# Buddy Algorithms

D. J. CHALLAB AND J. D. ROBERTS
*Department of Computer Science, University of Reading, Whiteknights, Reading RG6 2AH*

***Algorithms are derived for allocating and deallocating blocks of memory using Binary Buddy, Weighted Buddy and Variant Weighted Buddy partitioning; and transformations are demonstrated for converting recursive into non-recursive forms. Algorithms are given for the relatively difficult computation of the addressed and size of a Buddy slot in the Weighted versions, and complexity and optimisation are also considered.***

*Received January 1986, revised May 1986*

## 1. INTRODUCTION

Outline descriptions of various forms of Buddy memory-management algorithms may be found in the scientific literature,[1,6,7,9] but considerable detailed design is required of any investigator wishing to make use of these. It is hoped that the time-consuming effort involved in such design will be unnecessary as a result of the following precise description. Substantially equivalent algorithms have been used[4,5] in several application experiments which demonstrate the use of Buddy storage management for implementing flexible arrays.

The original Binary Buddy System,[6,7] working in a memory of size $2^n$, allows users to request a memory slot of size $2^k$ for a suitable range of $k$. For consistency of discussion it is necessary to define a smallest size of slot even though this decision is arbitrary and of no fundamental significance. To allow $k = 0$ would allow a slot to be a single word; we assume $1 \leqslant k \leqslant n$ to reflect two practical considerations, namely: (i) it is common to use two-way linked lists in the storage management which requires two parts per slot; and (ii) the information in a slot of size 1 could as well be held in the space allocated for the pointers to such a slot.

The Weighted Buddy System allows a memory slot to be requested either of size $2^k$ where $1 \leqslant k \leqslant$ n or size $3.2^k$ where $1 \leqslant k \leqslant n-2$. The Variant buddy system[1] is similar to the Weighed buddy system except that when a slot of size $2^{k+2}$ fragments successively into slots of sizes $3.2^k$ & $2^k$ and then $2^{k+1}$, $2^k$ & $2^k$, the two slots of size $2^k$ may be merged into a single slot of size $2^{k+1}$. These slots are not Buddies in the original partition process and a free slot of size $2^k$ may thus be merged in a non-unique way to form a larger free slot. The creation and deletion of a slot of memory in either system can be described by the pair of commands

*new* $(p, t)$ which returns the address of a new slot in $p$, and

*dispose* $(p, t)$ which frees the slot at $p$ for future use.

Here $p$ is a variable of type *address* and $t$ is of type *tag* (an integer sub-range) which takes even values for slot in the Binary Buddy system. We define *size*$[t]$ as follows

$$size[2k] = 2.2^k \, (k \geqslant 1)$$

$$size[2k+1] = 3.2^k \, (k \geqslant 1)$$

$$size[0] = 2$$

Note that the value 1 is not used as a tag.

The purpose of this paper is to study the design of algorithms for implementing *new* $(p, t)$ and *dispose* $(p, t)$. These are presented firstly as recursive algorithms and secondly in their more practicable non-recursive form.

Although the Binary Buddy and Weighted Buddy allocation algorithms are different, they have the same structure and can be unified into a common form by postponing the definition of certain functions which determine the address and size of the Buddy of any given slot. These functions are:

| | |
|---|---|
| *Larger* $(t)$ | the tag for the next larger size, |
| $B_p(p, t), B_t(p, t)$ | the address and size tag respectively of the Buddy of the slot at $p$ with size tag $t$, and |
| $P_p(p, t), P_t(p, t)$ | the address and size tag respectively of the parent of the slot at $p$ with size tag $t$. |

We note that

$$P_p(p, t) = \min(p, B_p(p, t)).$$

It is assumed that an array called *free* is declared of which each element *free* $[t]$ contains some representation of the free set of slots of size *size* $[t]$. For the algorithm to work, in fact for the indexing of *free* to remain in range, it is necessary that at least one non-empty set of free slots should exist of size at least *size*$[t]$. The implication of this will be discussed later, but for the moment we just assume that the responsibility for this rests with the writing of any calling program.

## 2. RECURSIVE FORM

The description to be given of the storage-management algorithms assumes that addressing begins at location 0 and is assumed to be within the scope of the type declaration

```
type address = 0..$2^{n-1}$;
     tag = 0..n-1;
```

The following pair of procedures describe the Binary and Weighted forms in a unified way dependent on definitions of certain functions.

```
procedure new (var p : address ; t : tag);
{called only if a slot exists with size tag $\geqslant t$}
  procedure newp (t : tag);
  begin
    if not empty (free [t])
    then begin
        'set $p$ to any member of $free\ [t]$';
        'remove ($p$) from $free\ [t]$'
        end
    else begin
     {since $free\ [t]$ is empty there must be a non-empty}
     {set of slots of at least the next larger size up.   }
        $newp\ (Larger\ (t))$;
        'insert ($B_p(p, t)$*) into $free\ [B_t(p, t)]$'
        end

  end;
begin
  $newp\ (t)$
end

procedure $dispose\ (p{:}address; t{:}tag)$;
var $present{:}boolean$;
begin
  'test if $B_p(p, t)$ is present in $free\ [B_t(p, t)]$';
  if not $present$
  then 'insert ($p$) into $free\ [t]$'
  else begin
       'remove ($B_p(p, t)$) from $free\ [B_t(p, t)]$';
       $dispose\ (P_p(p, t),\ P_t(p, t))$
       end;
end
```

In all the above we can replace the functions $B_p$, $B_t$, $P_p$ and $P_t$ either by the Binary Buddy functions which we call $BB_p$, $BB_t$, $BP_p$ and $BP_t$ or with the corresponding functions for the Weighted Buddy system which we call $WB_p$, $WB_t$, $WP_p$ and $WP_t$. The Binary Buddy functions can be written down quite simply as

$$Larger(t) = t+2$$

$$BB_p(p,t) = \begin{cases} p+size[t] & when\ p/size[t]\ is\ even \\ p-size[t] & when\ p/size[t]\ is\ odd \end{cases}$$

$$BB_t(p, t) = t$$

$$BP_p(p, t) = size[t+2]^* \lfloor p\ /\ size[t+2] \rfloor$$

$$BP_t(p, t) = t+2$$

The Weighted Buddy functions are more complex. Their defintion is considered in Section 4.

With the Variant Weighted form the form of procedure *new* is exactly as for the Weighted Buddy, but the dispose procedure, called *Vdispose*, is of a more complex form and uses both sets of functions. Here, we look to combine the slot with either a Weighted Buddy or Binary Buddy slot and not just a Weighted Buddy as in the ordinary Weighted version.

```
procedure $Vdispose\ (p{:}address; t{:}tag)$;
var $bpresent,\ wpresent{:}boolean$;
begin
  'test if $BB_p(p, t)$ is present in $free\ [BB_t(p, t)]$';
  'test if $WB_p(p, t)$ is present in $free\ [WB_t(p, t)]$';
  if $bpresent$ then begin
          'remove ($BB_p(p, t)$) from $free\ [BB_t(p, t)]$';
               $Vdispose\ (BP_p(p, t),\ BP_t\ (p, t))$
        end else
  if $wpresent$ then begin
          'remove ($WB_p(p, t)$) from $free\ [WB_t(p, t)]$';
               $Vdispose\ (WP_p(p, t),\ WP_t\ (p, t))$
        end
  else 'insert ($p$) into $free\ [t]$'
end;
```

* Since $B_p(p, t)$ is the right Buddy we can replace it by $p+size[t]$.

All forms of procedure *new* may be called only if there is a non-empty set of slots of at least the size requested. It is important to be able to ascertain whether this is true, and a convenient way to do this is to maintain an integer variable called *freemax*, which is the largest integer $i$ for which $free[i]$ is a non-empty set, or -1 if no such $i$ exists.

After each call of *new* $(p, t)$, *freemax* either stays the same or decreases. This is because the insert operation is called only after a larger slot has already been taken from a non-empty free list with higher tag value than that into which the new slot is inserted. The important thing about *freemax* is that is removes the need for a failure exit when no slot exists, because it provides a way for the user to test quickly if a slot does exist. When procedure *new* is called, *freemax* may decrease but it will certainly not increase. Therefore we can ensure that *freemax* is always maintained to its true value for the user if at the end of every call of *new* we insert an operation to update *freemax* downwards at the end of each user call (but not necessarily in the recursive calls) of $new(p, t)$, namely:

```
while $empty\ (free\ [freemax])$ and $freemax \geqslant 0$
do $freemax{:} = freemax\text{-}1$;
```

Conversely *freemax* can increase but will certainly not decrease when procedure *dispose* is called. If it does increase, it does so at the point in the procedure where $free[t]$ changes from being empty to non-empty. Thus *freemax* can be updated by inserting the statement

```
if $t > freemax$ then $freemax{:} = t$
```

after the call of 'insert ($p$) into $free\ [t]$'.

Note that to avoid the possibility of an out-of-range subscript without resorting to a non-symmetric definition of the **and** in the **while** condition above, it is convenient to declare

```
var $free$: array $[-1\,..\,2n]$ of slotset;
                              with $free[-1] = empty$.
```

## 3. NON-RECURSIVE FORMS

Although recursive procedures can often be particularly clear and readily provable, it is often desirable to represent algorithms in non-recursive form on actual machines (cf. Knuth[8] and Bird[2, 3]). We therefore need to consider how recursive procedures can be converted to non-recursive ones generally. Often one can eliminate mechanically the last call a procedure makes to itself. If a procedure *proc* $(p, i)$ (where $i$ is a value parameter) has a call to *proc* $(p,j)$ as its last step we can replace the call *proc* $(p,j)$ by an assignment, $i{:} = j$, followed by a jump to the beginning of the code for procedure *proc*. This kind of elimination is called 'tail recursion' elimination. The recursion in the *dispose* procedure is of this kind but the procedure *new* is not, and in such cases a more general and less simple approach is required.

Conversion to non-recursive form involves two phases which we consider in turn. The first stage is to transform the algorithms into forms in which the recursive parts are free of parameters and local variables. The second stage is to express the recursive parts in non-recursive form.

The principles of the second stage of the technique are: (1) to identify the structure of the recursive parts; (2) to state which pieces of program correspond to the 'boxes' in the structure; and (3) to transform the structure, substitute back the pieces of program and perform any final simplifications.

When the first stage is done and when the code for updating *freemax* is inserted we have the following.

```
procedure new (var p: address; t_0: tag);
var t: tag;
  procedure new_pt;
  begin
    if not empty (free[t])
    then begin
           'set p to any member of free[t]';
           'remove (p) from free[t]'
         end
    else begin
           t: = Larger (t); new_pt;
           t: = Smaller (t);
           'insert (p+size[t]) into free[B_t(p, t)]'
         end
  end;
begin
  t: = t_0;
  new_pt;
  while empty (free[freemax]) and freemax > 0
  do freemax: = freemax - 1
end
```

Here all functions are as defined in the earlier form except for *Smaller*($t$), which is the inverse function of *Larger*($t$).

Procedure $new_{pt}$ is of the form

```
if b then A
  else begin
    B;
    new_pt;
    C
    end
```

In general, elimination of a single occurrence of recursion in an algorithm requires construction of a variable called *level* to record the level of recursion. So procedure $new_{pt}$ becomes

```
begin
  level: = 0;
  while not b do
    begin
      B;
      level: = level+1
    end;
  A;
  while level > 0 do
    begin
      level: = level-1;
      C
    end
end
```

In this case *level* is redundant, as equivalent information is contained in $t$, the incrementing of *level* always being coupled with $t := Larger(t)$ and the decrementing of *level* with $t := Smaller(t)$. Thus

$t = Larger^{(level)}(t_0)$  {*level* applications of $Larger(t_0)$}

The test '$level > 0$' can therefore be adequately replaced by $t > t_0$.

The following procedure represents the algorithm $new_{pt}$ in a non-recursive form.

```
procedure new_pt;
begin
  while empty (free[t]) do t: = Larger(t);
  'set p to any member of free[t]';
  'remove (p) from free[t]';
  while t > t_0 do
    begin
      t: = Smaller(t);
      'insert (p+size(t)) into free[B_t(p, t)]'
    end;
end
```

The final form of algorithm *new*, after substituting the non-recursive form of $new_{pt}$, is thus as follows.

```
procedure new (var p: address; t_0: tag);
var t: tag;
begin
  t: = t_0;
  while empty (free[t]) do t: = Larger(t);
  'set p to any member of free [t]';
  'remove (p) from free [t]';
  while t > t_0 do
  begin
    t: = Smaller(t);
    'insert (p+size[t]) into free [B_t(p, t)]'
  end;
while empty (free[freemax]) and freemax > 0
    do freemax: = freemax-1;
end
```

The following procedure represents the first stage of the transformation of *dispose* (with the inclusion of the statement for updating *freemax*) in which the recursive part is parameter-free.

```
procedure dispose (p_0: address; t_0: tag);
var p: address; t: tag; present: boolean;
  procedure dispose_pt;
  begin
    'test if B_p(p, t) is present in free [B_t(p, t)]';
    if not present
    then begin
           if t > freemax then freemax: = t;
           'insert (p) into free [t]'
         end
    else begin
           'remove (B_p(p, t)) from free [B_t(p, t)]';
           p: = P_p(p, t); t: = P_t(p, t);
           dispose_pt
         end
  end;
begin
  p: = p_0; t: = t_0;
  dispose_pt
end
```

The recursive part has the following non-recursive form.

```
procedure dispose_pt;
begin
  'test if B_p(p, t) is present in free [B_t(p, t)]';
```

```
    while present do
      begin
        'remove ($B_p(p, t)$) from free [$B_t(p, t)$]';
        $p := P_p(p, t); t := P_t(p, t)$;
        'test if $B_p(p, t)$ is present in free [$B_t(p, t)$]'
      end;
    if $t > freemax$ then $freemax := t$;
    'insert ($p$) into free[$t$]'
end
```

The following final form of *dispose* is obtained by the substitution of the non-recursive form of $dispose_{pt}$.

```
procedure dispose ($p_0$: address; $t_0$: tag);
var p: address; t: tag; present: boolean;
begin
    $p := p_0; t := t_0$;
    'test if $B_p(p, t)$ is present in free [$B_t(p, t)$]';
    while present do
      begin
        'remove ($B_p(p, t)$) from free [$B_t(p, t)$]';
        $p := P_p(p, t); t := P_t(p, t)$;
        'test if $B_p(p, t)$ is present in free [$B_t(p, t)$]'
      end;
    if $t > freemax$ then $freemax := t$;
    'insert ($p$) into free[$t$]'
end
```

To express the Variant Weighted algorithm in non-recursive form the *dispose* algorithm has to be converted first into recursive form with formal parameters and finally into non-recursive form without formal parameters.

```
procedure Vdispose ($p_0$: address; $t_0$: tag);
var p: address; t: tag;
    procedure $Vdispose_{pt}$;
    begin
      'test if $BB_p(p, t)$ is present in free [$BB_t(p, t)$]';
      'test if $WB_p(p, t)$ is preent in free [$WB_t(p, t)$]';
      if bpresent then begin
                          'remove ($BB_p(p, t)$) from free
                          [$BB_t(p, t)$]';
                          $p := BP_p(p, t); t := BP_t(p, t)$;
                          $Vdispose_{pt}$
                        end else
      if wpresent then begin
                          'remove ($WB_p(p, t)$) from free
                          [$WB_t(p, t)$]';
                          $p := WP_p(p, t); t := WP_t(p, t)$;
                          $Vdispose_{pt}$
                        end
      else 'insert ($p$) into free[$t$]';
    end;
begin
    $p := p_0; t := t_0$;
    $Vdispose_{pt}$
end
```

Procedure $Vdispose_{pt}$ is of the form

```
A;
if bpresent
  then begin
         B;
         $Vdispose_{pt}$
  end else
if wpresent
  then begin
         C;
         $Vdispose_{pt}$
    end
else D;
```

On converting to a non-recursive form procedure $Vdispose_{pt}$ becomes

```
A;
while bpresent or wpresent do
  begin
    if bpresent
    then B
    else C;
    A
  end;
D;
```

Using the same technique as before, we can substitute the non-recursive form of procedure $Vdispose_{pt}$ into procedure *Vdispose* to get the final non-recursive form of the Variant Weighted Buddy algorithm.

```
procedure Vdispose ($p_0$: address; $t_0$: tag);
var p: address; t: tag; bpresent, wpresent: boolean;
begin
    $p := p_0; t := t_0$;
    'test if $BB_p(p, t)$ is present in free [$BB_t(p, t)$]';
    'test if $WB_p(p, t)$ is present in free [$WB_t(p, t)$]';
    while wpresent or bpresent do
      begin
        if wpresent then begin
                            'remove ($p$) from free[$t$]';
                            $p := WP_p(p, t); t := WP_t(p, t)$
                          end
                   else begin {bpresent}
                            'remove ($BB_p(p, t)$) from free
                            [$BB_t(p, t)$]';
                            $p := BP_p(p, t); t := BP_t(p, t)$
                          end;
        'test if $BB_p(p, t)$ is present in free [$BB_t(p, t)$]';
        'test if $WB_p(p, t)$ is present in free [$WB_t(p, t)$]'
      end;
    if $t > freemax$ then $freemax := t$;
    'insert ($p$) into free[$t$]'
end
```

## 4. THE DIFFICULT FUNCTIONS

In the Binary Buddy partition, the size of every slot is given by $2^{(t/2)+1}$ and it is easy to ascertain whether a slot at address $p$ is a left buddy or right buddy from the ratio defined by $R = (p/2^{(t/2)+1})$ (which is always an integer), (proved in Appendix 2). The slot is a left buddy if $R$ is even and a right buddy if $R$ is odd.

In the Weighted Buddy partition, it is not in general nearly so easy to ascertain how any given slot has been generated. Omitting first the special case of the Binary partition of free slots of size 4, four cases need to be considered. A slot at $p$ of size $2^k$ can be

(*a*) (when $k > 1$) the left child of a slot of size $3.2^{k-1}$ at $p$ and the left–left grandchild of a slot of size $2^{k+1}$ at $p$, or

(*b*) the right child of a slot of size $3.2^k$ at $p-2^{k+1}$ and the left–right grandchild of a slot of size $2^{k+2}$ at $p-2^{k+1}$, or (*c*) the right child of a slot of size $2^{k+2}$ at $p-3.2^k$; while a slot at $p$ of size $3.2^k$ can only be

(*d*) the left child of a slot of size $2^{k+2}$ at $p$, and therefore the 3:1 splitting of a slot of size $2^{k+2}$ is the only way that a slot of size divisible by 3 is generated.

With the special Binary partition case, a slot at $p$ of size 2 exactly can also be

(*e*) the left child of a slot of size 4 at $p$, or
(*f*) the right child of a slot of size 4 at $p-2$.

If a slot is of size $3.2^k$ it must be of case (*d*). If a slot is of size $2^k$, the case is not uniquely determined by the size alone and it is useful to consider the properties of the ratio of the parent or grandparent. Suppose the slot in question has ratio $R$, then in case (*a*) the ratio of the grandparent is $R/2$.

Case (*a*) can therefore apply only if $R$ is even, i.e.

$$R \textbf{ mod } 4 \in \{0, 2\}$$

In case (*b*) the ratio of the grandparent is $[(R-2)/4]$, therefore

$$R \textbf{ mod } 4 = 2$$

In case (*c*) the ratio of the parent is $[(R-3)/4]$, therefore

$$R \textbf{ mod } 4 = 3$$

In case (*e*) the ratio of the parent is $R/2$, therefore

$$R \textbf{ mod } 4 \in \{0, 2\}$$

and in case (*f*) it is $[R-1)/2]$, therefore

$$R \textbf{ mod } 4 \in \{1,3\}$$

Thus the origin of a slot of size $2^k$ is indicated by $R$ **mod** 4 when this takes the value 0 or, 3, but when $R$ **mod** 4 = 2 in general more analysis is required. To proceed further we need the following definition and lemma.

## Definition

The boolean function $ex(R)$, where $R$ is any positive integer, is defined to take the value *true* if and only if $R+1$ can be expressed in the form $(2n+1)*4^r$, i.e., if and only if 2 is a factor of $R+1$ with even multiplicity.

## Lemma

A slot can be generated in cases (*a*), (*b*) or (*c*) with ratio $R$ only if $ex(R)$ takes the value *true* (proof in Appendix 1).

Using this lemma we can resolve between the cases (*a*) and (*b*), because in case (*a*) a grandparent exists with ratio $R/2$ so that the factor 2 has even multiplicity in $(R/2+1)$ and in case (*b*) a grandparent exists with ratio $R-2/4$ so that the factor 2 has even multiplicity in $(R-2/4+1) = \frac{1}{2}(R/2+1)$. If the multiplicity of 2 is odd in $R/2+1$, case (*a*) is eliminated; and if the multiplicity of 2 is even in $R+1$, then it is odd in $\frac{1}{2}(R/2+1)$ thus eliminating case (*b*). Thus we can distinguish between cases (*a*) and (*b*) by whether the boolean function $ex(R/2)$ takes the value *true* or *false* respectively. To trace the origin of the slot we can therefore refer to Table 1*a*.

To implement the procedure *new*, it is sufficient to compute the function $WB_t(p, t)$. The algorithm for this is obtained from Table 1 as follows.

**Table 1. The 'difficult' functions in the Weighted Buddy system: (*a*) identification of cases; (*b*) values of functions**
**Table 1(*a*)**

| | *t* even | | |
|---|---|---|---|
| $R$ **mod** 4 | $t = 0$ | $t > 1$ | *t* odd |
| 0 | Case e | Case a | |
| 1 | Case f | IMPOSSIBLE | Case d |
| 2 | $ex\left(\frac{R}{2}\right) \rightarrow$ case b | | |
| | NOT $ex\left(\frac{R}{2}\right) \rightarrow$ case a | | |
| 3 | $ex(R) \rightarrow$ case c | | |
| | NOT $ex(R) \rightarrow$ case f | Case c | |

For definition of $ex(R)$ see Section 4.

**Table 1(*b*)**

| Case | $WP_p(p, t)$ | $WP_t(p, t)$ | $WB_p(p, t)$ | $WB_t(p, t)$ | *size*[*t*] |
|---|---|---|---|---|---|
| *a* | $p$ | $t+1$ | $p+2^{(t/2)+1}$ | $t-2$ | $2^{(t/2)+1}$ |
| *b* | $p-2.2^{(t/2)+1}$ | $t+3$ | $p-2.2^{(t/2)+1}$ | $t+2$ | $2^{(t/2)+1}$ |
| *c* | $p-3.2^{(t/2)+1}$ | $t+4$ | $p-3.2^{(t/2)+1}$ | $t+3$ | $2^{(t/2)+1}$ |
| *d* | $p$ | $t+1$ | $p+3.2^{(t-1)/2}$ | $t-3$ | $3.2^{(t-1)/2}$ |
| *e* | $p$ | 2 | $p+2$ | 0 | 2 |
| *f* | $p-2$ | 2 | $p-2$ | 0 | 2 |

```
function $WB_t(p: address; t: tag): tag$;
var $R: integer$;
begin
  if ($t$ mod 2) = 0
    then begin
          $R := \frac{p}{size[t]}$;
          case ($R$ mod 4) of
            0: $WB_t := t-2$;
            1: $WB_t := 0$;
            2: if $ex\left(\frac{R}{2}\right)$
                then $WB_t := t+2$
                else $WB_t := t-2$;
            3: if $t > 0$
                then $WB_t := t+3$
                else if $ex(R)$
                then $WB_t := 3$
                else $WB_t := 0$
          end
        end
    else $WB_t := t-3$
end
```

This is satisfactory for procedure *new*. But in *dispose* the analysis of cases is best introduced by restructuring the whole algorithm.

The reasons for procedure *Wdispose* needing restructuring are (*a*) case identification is not trivial (in some cases *ex* (*R*) is involved); (*b*) up to four function calls may be needed, all for the same case of the pair $(p, t)$.

In the procedure *dispose*, the values of $B_p(p, t)$, $B_t(p, t)$, $P_p(p, t)$ and $P_t(p, t)$ are computed in sequence, all for the same cases $(a, b, c, d, e, f)$. As these are in different parts of the loop and, as identification of the case is a major part of the computation, it is useful to unbundle the identification of the case from the evaluation of the functions. Thus, after each pair of assignments to $p$ and $t$ we identify the case, as in the following restructured form of the non-recursive form of the Weighted Buddy procedure *Wdispose*.

```
procedure Wdispose ($p_0$: address; $t_0$: tag);
var p: address; t: tag; present: boolean;
  fcase: (a, b, c, d, e, f);
  bp: address; bt: tag;
begin
  $p := p_0; t := t_0$;
  'identity fcase';
  'compute bp, bt from fcase, p, t';
  'test if bp is present in free [bt]';
  while present do
    begin
      'remove (bp) from free[bt]';
      '$p := WP_p(p, t); t := WP_t(p, t)$ given fcase';
      'identity fcase';
      'compute bp, bt from fcase, p, t';
      'test if bp is present in free [bt]'
    end;
  if $t >$ freemax then freemax := t;
  'insert (p) into free[t]'
end
```

It remains to express the algorithms for computing parent and Buddy slots which is again done with reference to Table 1(*b*).

```
'identify fcase':
if (t mod 2) = 0
then begin
        $R := \frac{p}{size[t]}$;
        case (R mod 4) of
          0: if $t > 0$
             then fcase := a
             else fcase := e;
          1: fcase := f;
          2: if $ex\left(\frac{R}{2}\right)$
             then fcase := b
             else fcase := a;
          3: if $t > 0$
             then fcase := c
             else if ex (R)
               then fcase := c
               else fcase := f
        end
     end
else fcase := d
```

```
'$p := P_p(p, t)$; $t := P_t(p, t)$ given fcase':
case fcase of
  a, d: $t := t + 1$;
  b: begin
       $p := p - size[t+2]$; $t := t + 3$
     end;
  c: begin
       $p := p - 3 * size[t]$; $t := t + 4$
     end;
  e: $t := 2$;
  f: begin
       $p := p - 2$; $t := 2$
     end
end
```

```
'compute bp, bt from fcase, p, t':
case f case of
  a: begin
       $bp := p + size[t]$; $bt := t - 2$;
     end;
  b: begin
       $bp := p - size[t+2]$; $bt := t + 2$
     end;
  c: begin
       $bp := p - size[t+3]$; $bt := t + 3$
     end;
  d: begin
       $bp := p + size[t]$; $bt := t - 3$
     end;
  e: begin
       $bp := p + 2$; $bt := 0$
     end;
  f: begin
       $bp := p - 2$; $bt := 0$
     end
end
```

## 5. COMPLEXITY CONSIDERATIONS

To make a quantitative measure of the performance of the procedures *new* and *dispose*, we need to consider the number of remove, insert and test operations, the numbers of calculations of $B_p$, $B_t$, $P_p$, $P_t$ and the number of miscellaneous basic operations.* This measure captures the essence of the computation while at the same time being divorced from any particular situation. There is no fundamental difference in terms of complexity between the recursive and the non-recursive forms of the algorithms (except that the refinement using *freemax* has been applied only to the non-recursive versions). Calculation of $B_p$ in the Binary Buddy system is straightforward and $B_t$ is trivial, $BB_t \equiv t$, but both these functions are more difficult for the Weighted Buddy

**Table 2. Analysis of complexity of *new*, *dispose* and *Vdispose***

| | *Bnew* | *Bdispose* | *Wnew and Vnew* | *Wdispose* | *Vdispose* |
|---|---|---|---|---|---|
| 'Insert' | $\lambda$ | 1 | $\lambda$ | 1 | 1 |
| 'Remove' | 1 | $\mu$ | 1 | $\mu$ | $\mu$ |
| 'Test' | — | $1+\mu$ | — | $1+\mu$ | $2+\mu$ to $2+2\mu$ |
| Calculation of $B_p, B_t$ $P_p, P_t$ | — | $1+2\mu$ | $\lambda$ | $1+2\mu$ | $1+2\mu$ |

* These are: indexing of an array, addition, subtraction, multiplication, assignment to variable, incrementing and decrementing.

**Table 3. Combined complexity of different variations of *new* and *dispose***

| | *Bnew* $(p, t)$ *and Bdispose* $(p, t)$ | *Wnew* $(p, t)$ *and Wdispose* $(p, t)$ | *Vnew* $(p, t)$ *and Vdispose* $(p, t)$ |
|---|---|---|---|
| 'Insert' + 'remove' + 'test' | $\lambda+1$ | $\lambda+1$ | $\lambda+1$ |
| Extra tests | — | — | Between 1 and $\mu+1$ |
| Calculation of the functions | $1+2\lambda$ Binary functions | $1+3\lambda$ ('Difficult' functions of Section 4) | $1+3\lambda$ (Mixed) |

system. Table 2 illustrates the complexity of the various operations involved in *new* and *dispose* and the Variant Weighted version *Vdispose*, the figures being expressed in terms of the average length of search $\lambda$ required to find a non-empty free list and the average number $\mu$ of 'remove' operations caused by combining a newly disposed slot with existing free slots. The actual quantitative performance of any of the Buddy systems being considered is liable to be dependent both on the characteristics of the machine being used and on the distribution of the request sizes. A full experimental investigation is beyond the scope of this paper, but elsewhere one of the authors[5] has observed $\lambda$ (and $\mu$) to take values between 1.53 and 2.02 in an application requiring fairly random dynamic allocation of slots of several different sizes. We can show how from theoretical consideration it is possible to make certain partial qualitative comparisons. In a long run, the number of slots removed must balance the number inserted and the numbers of calls of *new* and *dispose* must also balance. Thus it follows from theoretical arguments that $\lambda = \mu$ and its also follows that the total memory management overhead is indicated by the cost of one call of *new* plus one call of *dispose*. This leads us to Table 3, points of which to note are the following.

(i) The Binary version involves evaluations of the Binary functions $BB_p$, etc. at the points where the Weighted version involves the functions $WB_p$, etc., which are the 'difficult' functions discussed in Section 4. There are also fewer Binary function calls because the function $BB_t(p, t) \equiv t$ and requires no computation at all.

(ii) The Variant Weighted version is likely to involve more executions of 'test' but fewer evaluations of the 'difficult' functions than the ordinary Weighted version, which differences will tend to cancel each other out.

The consequences in relation to implementation are that the memory-management operations can be kept down to the equivalent of a reasonable number of elementary operations provided all the set manipulation operations 'insert', 'remove' and 'test' can be reduced to a few elementary operations, and provided we can also similarly reduce the work involved in evaluating the functions $B_p$, $B_t$, $P_p$ and $P_t$. The operations 'insert' and 'remove' are simple to implement using standard techniques with two-way lists, but 'test', if it is to be efficient, requires the two-way list to be augmented by a bit-map which is also standard for Buddy systems. The functions $B_p$ etc. in the Binary version involve determining whether $p/size[t]$ is even or odd, which can be expressed as an inspection of bit $2t+1$ in $p$. Similarly, in the Weighted Buddy case the computation of *fcase* involves the assignment $R := p/size[t]$, but it is important to look at how $R$ is used. First, $R$ **mod** 4 is required (obtained by inspecting a pair of bits in $p$) to determine the case, and secondly, in the computation of $ex(R/2)$ or $ex(R)$, further pairs of bits in $p$ are inspected until at least one of the bits is zero. The assumption behind the estimates given is that the loop in function *ex* is executed an average $1\frac{1}{3}$ times, and there is an even mixture of the values of *fcase*.

## 6. CONCLUSION

Recursive and non-recursive forms of algorithms for the Binary Buddy, Weighted Buddy and Variant Weighted Buddy systems have been derived. The functions for finding the location and size of a Weighted buddy are more difficult than for a Binary buddy and generally require many iterations of basic operations.

The most efficient ways of computing these 'difficult' functions make use of bit indexing, which is a technique that is also used to optimise the testing for membership of slots of given size in free lists. With reasonable optimisation the mean computational complexity of the Weighted and Variant Weighted versions will still be more complex than that of the Binary version, but probably not by a large factor.

**Acknowledgement**

The authors wish to thank the painstaking but anonymous referee.

## REFERENCES

1. A. G. Bromley, An improved Buddy method for dynamic storage allocation. *Proc. 7th Aust. Comp. Conf.*, pp. 708–715 (1976).
2. R. S. Bird, Notes on recursion elimination. *Comm. of the ACM* **20** (6) (1977).
3. R. S. Bird, Recursion elimination with variable parameters. *The Computer Journal*, **22** (2) (1979).
4. D. J. Challab and J. D. Roberts, Flexible array implementation by Buddy methods. (Submitted for publication).
5. D. J. Challab, Ph.D. thesis (in preparation).

6. K. C. Knowlton, A fast storage allocator. *Comm. of the ACM* **8** (10) (1965), 623–625.
7. D. E. Knuth, *The Art of Computer Programming*. Vol. 1. *Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 2nd edition, pp. 111, 435–455. (1968).
8. D. E. Knuth, Structured programming with goto statements. *ACM Computing Surveys* **6** (4), 261–301 (1974).
9. K. K. Shen and J. L. Peterson, A weighted Buddy method for dynamic storage allocation. *Comm. of the ACM* **17** (10) 558–562 (1974); corrigendum **18** (4) 202 (1975).

## APPENDIX 1

### Lemma

A slot can be generated in the cases (*a*), (*b*) or (*c*) with ratio $R$ only if $ex(R)$ takes the value *true* (i.e. only if the factor 2 occurs in $R+1$ with even multiplicity).

### Proof (by induction)

Initially $R = 0$, therefore $R+1 = 1$, in which 2 has zero multiplicity. The rest of the proof is to show that evenness of multiplicity is preserved by the process of generating new slots. Any slot of size $2^k$ with ratio $R$ is one of the following (cf. Section 4):

(*a*) a left–left grandchild of a slot with ratio $R/2$;
(*b*) a left–right grandchild of a slot with ratio $R-2/4$;
(*c*) a right child of a slot with ratio $R-3/4$.

This means that the ratio $R_{new}$ of any newly generated slot (in case *a*, *b*, *c*) satisfies

$$R_{new} \in \{2R,\ 4R+2,\ 4R+3\}$$

whence

$$R_{new}+1 \in \{2(R+1)-1,\ 4(R+1)-1,\ 4(R+1)\}$$

Thus every value $\mathrm{R_{new}}+1$ produced must either be odd or 4 times a previous value, in both cases preserving the evenness of the multiplicity of 2 in $R_{new}+1$ (end of proof).

### Corollary 1

The quaternary representation of $R+1$ is of the form $\begin{Bmatrix}1\\3\end{Bmatrix}0^*$.

### Corollary 2

The quaternary representation of $R$ is of the form $\begin{Bmatrix}0\\2\end{Bmatrix}3^*$.

## APPENDIX 2

### Lemma

Blocks generated by Weighted splitting fall into two classes: **A** of size $2^k$ and address a multiple of $2^k$ ($k \geqslant 1$); **B** of size $3.2^k$ and address a multiple of $4.2^k$ ($k \geqslant 1$).

### Proof

The initial state of the memory is a single slot of size $2^m$ at location 0 (of class **A**). A subsequent Weighted splitting of a slot of class **A** of size $4.2^k$ located at $4R.2^k$ produces: (i) a slot of size $3.2^k$ at $R.2^k$ (which is of class **B**); and (ii) a slot of size $2^k$ at $(4R+3).2^k$ (which is of class **A**).

A slot of class **B** of size $3.2^k$ located at $4R.2^k$ splits into: (iii) a slot of size $2.2^k$ at $4R.2^k$ (which is of class **A**); and (iv) a slot of size $2^k$ at $(4R+2).2^k$ (which is of class **A**). In the special case of the Binary splitting a slot of size $2^2$ located at $R.2^2$ splits into slots of size 2 located at $4R$ and $4R+2$ (which are also both of class **A**) (end of proof).

### Corollary

The address of a slot of size $2^k$ (where $k \geqslant 1$) is a multiple of $2^k$.