

Chapter 9

Heap

Objectives

Upon completion you will be able to:

- Define and implement heap structures
- Understand the operation and use of the heap ADT
- Design and implement selection applications using a heap
- Design and implement priority queues using a heap

9-1 Basic Concepts

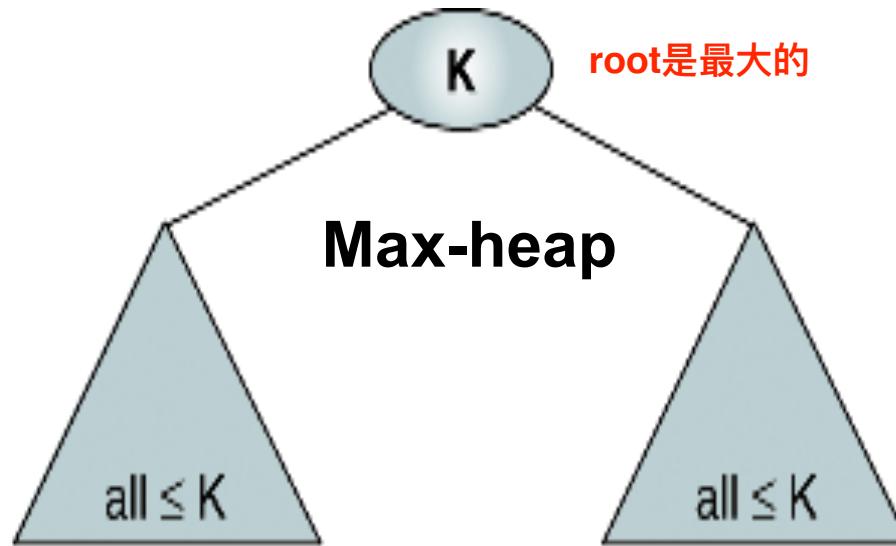
A heap is a binary tree whose left and right subtrees have values less than their parents. We begin with a discussion of the basic heap structure and its two primary operations, reheap up and reheap down.

- Definition
- Maintenance Operations

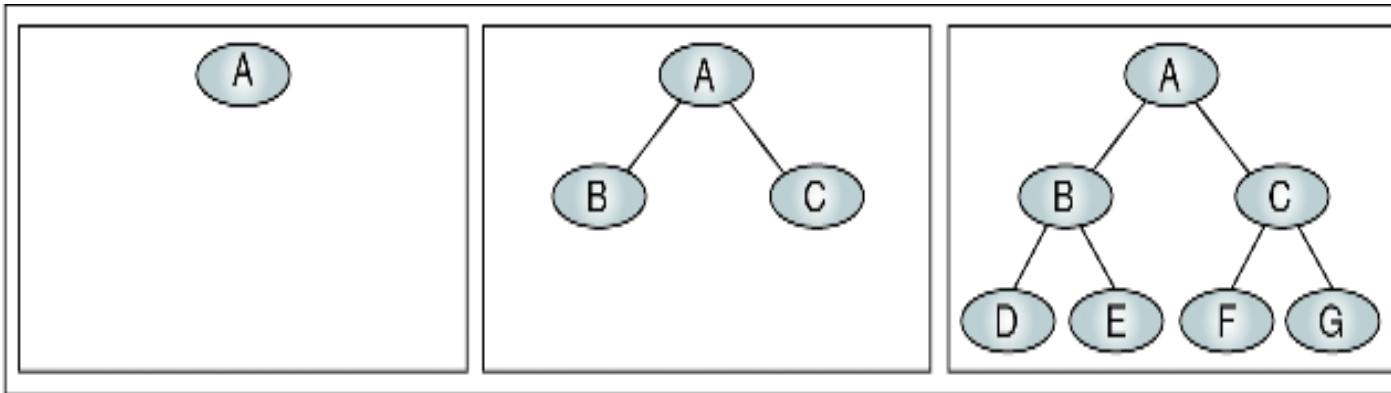
Heap

- Rule 1: the binary tree is complete or nearly complete
- Rule 2: the key value of each node is greater than or equal to the key value in each of its descendants
- A heap tree is often implemented in an array rather than a linked list which makes for very efficient processing
- It's meaningless to traverse the heap tree

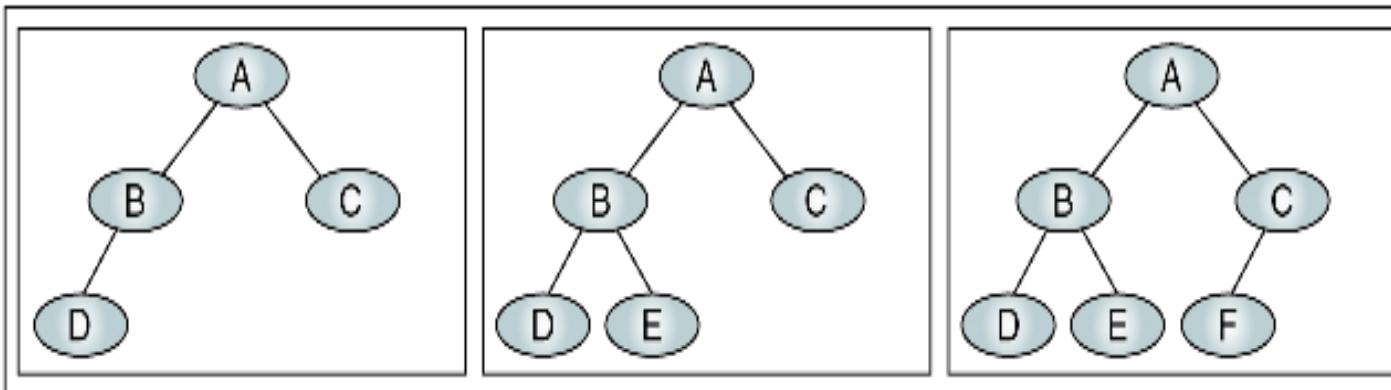
也有Min-heap
也就是root是最小的



Complete and Nearly Complete Trees

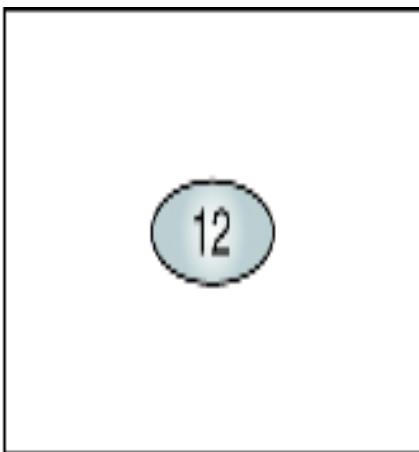


(a) Complete trees (at levels 0, 1, and 2)

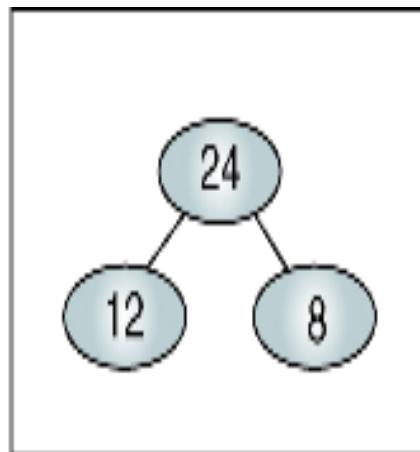


(b) Nearly complete trees (at level 2)

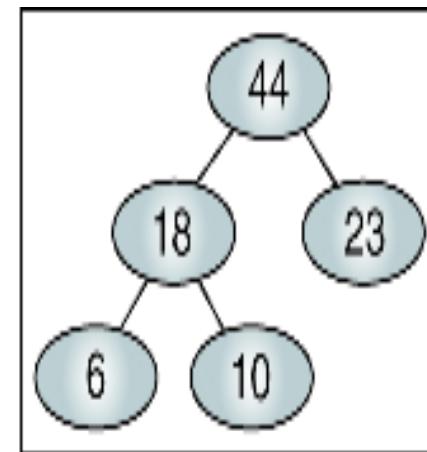
Heap Trees



(a) Root-only heap

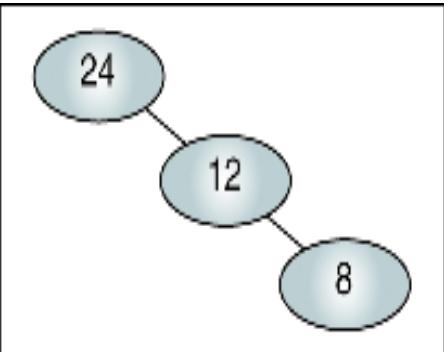


(b) Two-level heap

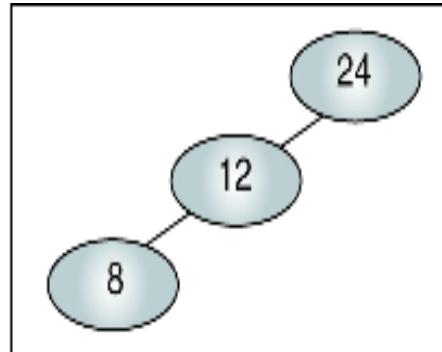


(c) Three-level heap
nearly complete

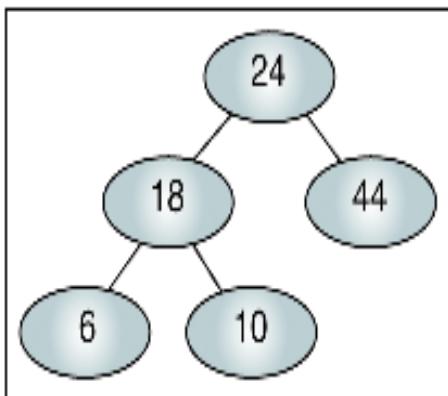
Invalid Heaps



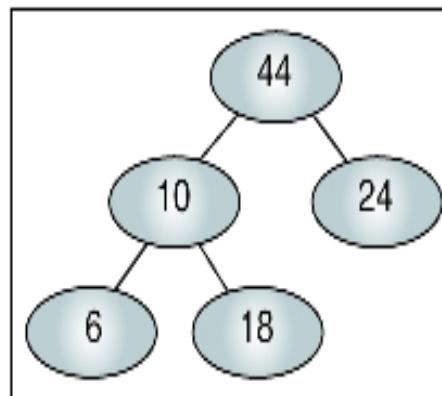
(a) Not nearly complete
(rule 1)



(b) Not nearly complete
(rule 1)



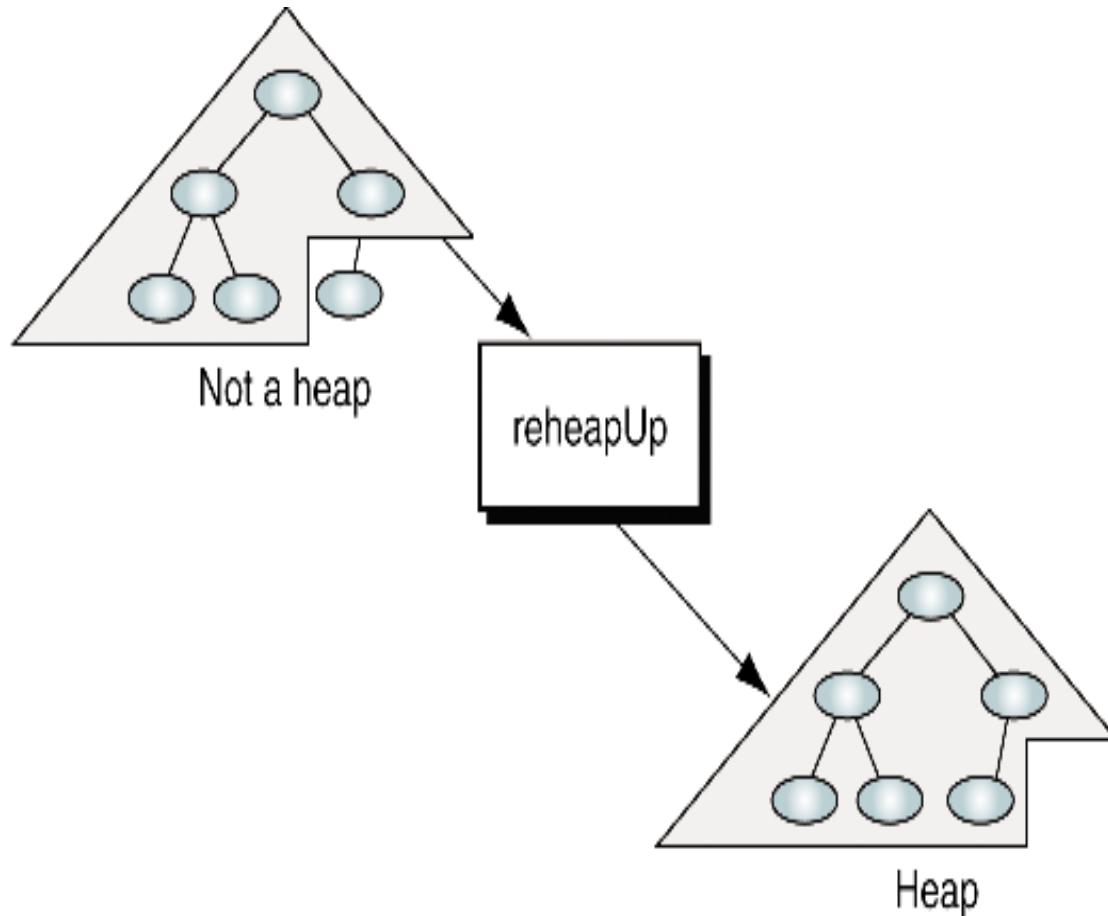
(c) Root not largest
(rule 2)



(d) Subtree 10 not a heap
(rule 2)

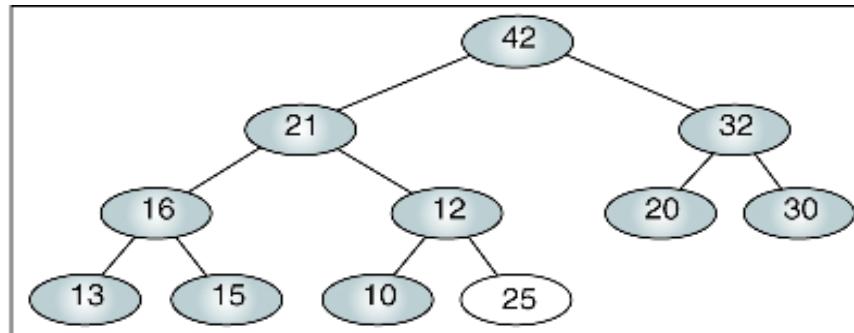
Reheap Up Operation

- After inserting a node into a heap, the reheap up operation may be needed

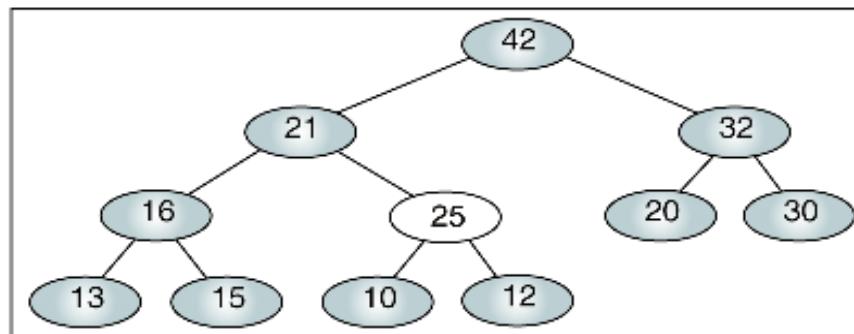


Reheap up Example

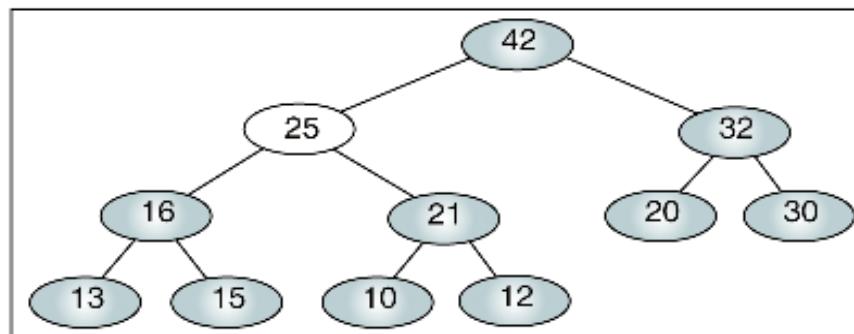
worst case: $O(\log n)$



(a) Original tree: not a heap



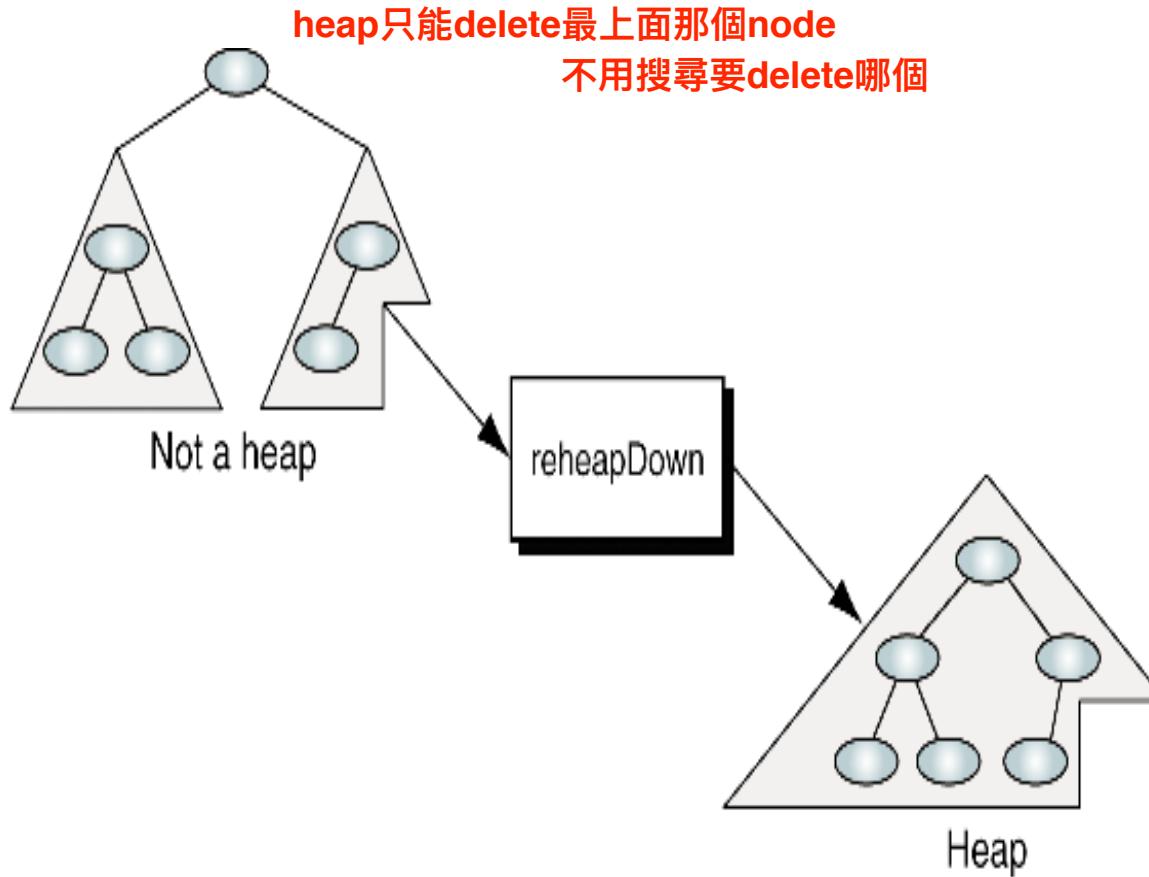
(b) Last element (25) moved up



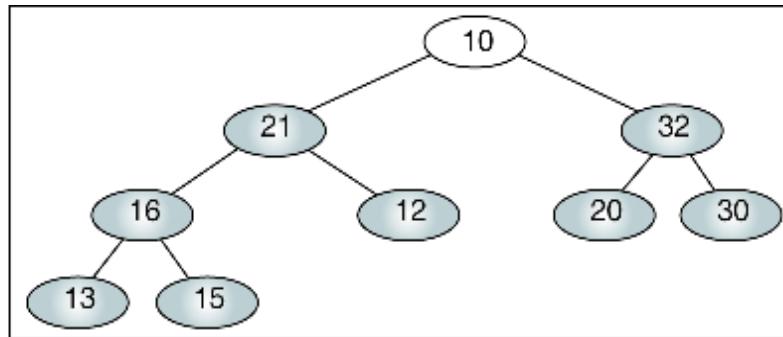
(c) Moved up again: tree is a heap

Reheap Down Operation

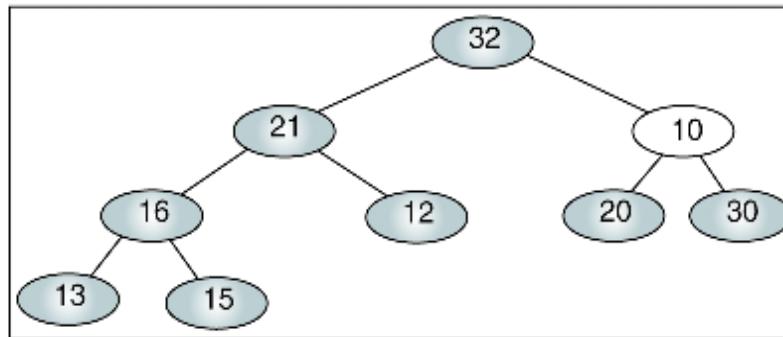
- After deleting a node (root), the reheap down is needed



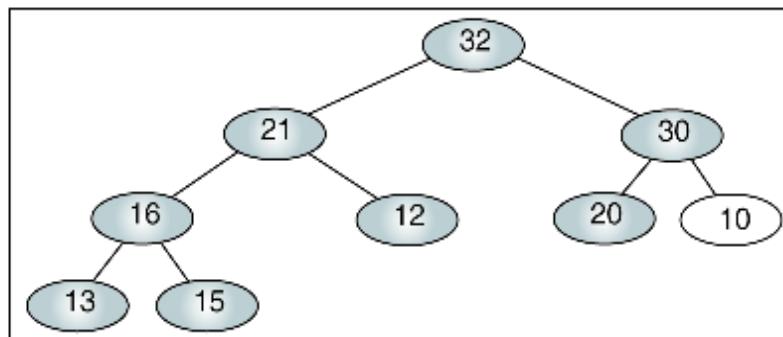
Reheap Down Example



(a) Original tree: not a heap



(b) Root moved down (right)



(c) Moved down again: tree is a heap

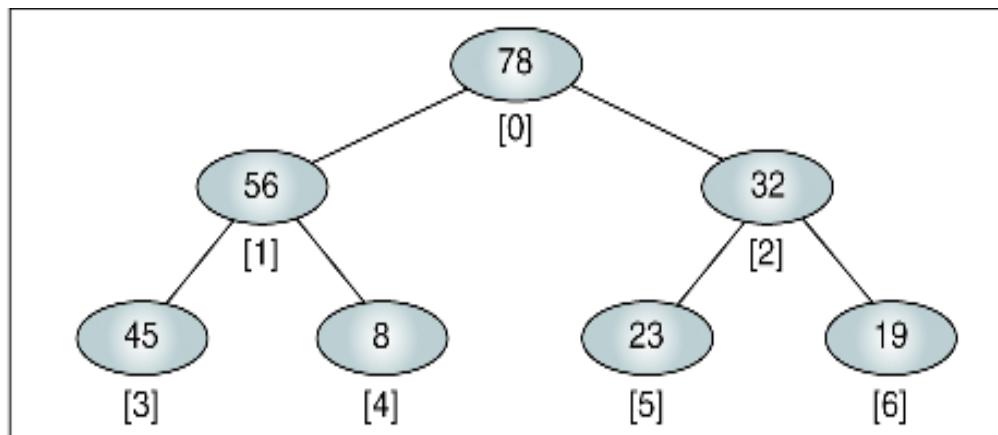
9-2 Heap Implementation

Heaps are usually implemented in an array structure. In this section we discuss and develop five heap algorithms.

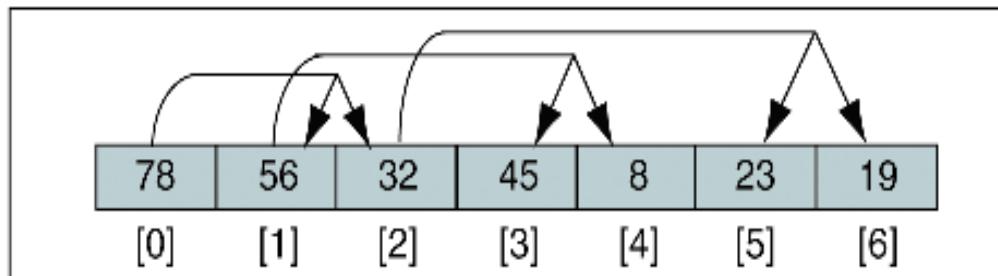
- Reheap Up
- Reheap Down
- Build a Heap
- Insert a Node into a Heap
- Delete a Node from a Heap

Heaps in Arrays

- For a node located at index i , its children are found at
 - Left child: $2i + 1$; right child: $2i + 2$; parent: $\lfloor (i-1)/2 \rfloor$ | $(i-1)/2$
- A heap can be implemented in an array because it must be a complete or nearly complete binary tree, which allows a fixed relationship between each node and its children



(a) Heap in its logical form



(b) Heap in an array

Reheap Up

```
Algorithm reheapUp (heap, newNode)
```

Reestablishes heap by moving data in child up to its
correct location in the heap array.

Pre heap is array containing an invalid heap

¬newNode is index location to new data in heap

Post heap has been reordered

```
1 if (newNode not the root)
    1 set parent to parent of newNode
    2 if (newNode key > parent key)
        1 exchange newNode and parent)
        2 reheapUp (heap, parent)
    3 end if
2 end if
end reheapUp
```

Reheap Down

Algorithm reheapDown (heap, root, last)

Reestablishes heap by moving data in root down to its correct location in the heap.

Pre heap is an array of data

 root is root of heap or subheap

 last is an index to the last element in heap

Post heap has been restored

Determine which child has larger key

1 if (there is a left subtree)

 1 set leftKey to left subtree key

 2 if (there is a right subtree)

 1 set rightKey to right subtree key

 3 else

 1 set rightKey to null key

Reheap Down (cont.)

```
4 end if
5 if (leftKey > rightKey)
  1 set largeSubtree to left subtree
6 else
  1 set largeSubtree to right subtree
7 end if
  Test if root > larger subtree
8 if (root key < largeSubtree key)
  1 exchange root and largeSubtree
  2 reheapDown (heap, largeSubtree, last)
9 end if
2 end if
end reheapDown
```

Build Heap

```
Algorithm buildHeap (heap, size)
```

Given an array, rearrange data so that they form a heap.

Pre heap is array containing data in nonheap order

 size is number of elements in array

Post array is now a heap

1 set walker to 1

2 loop (walker < size)

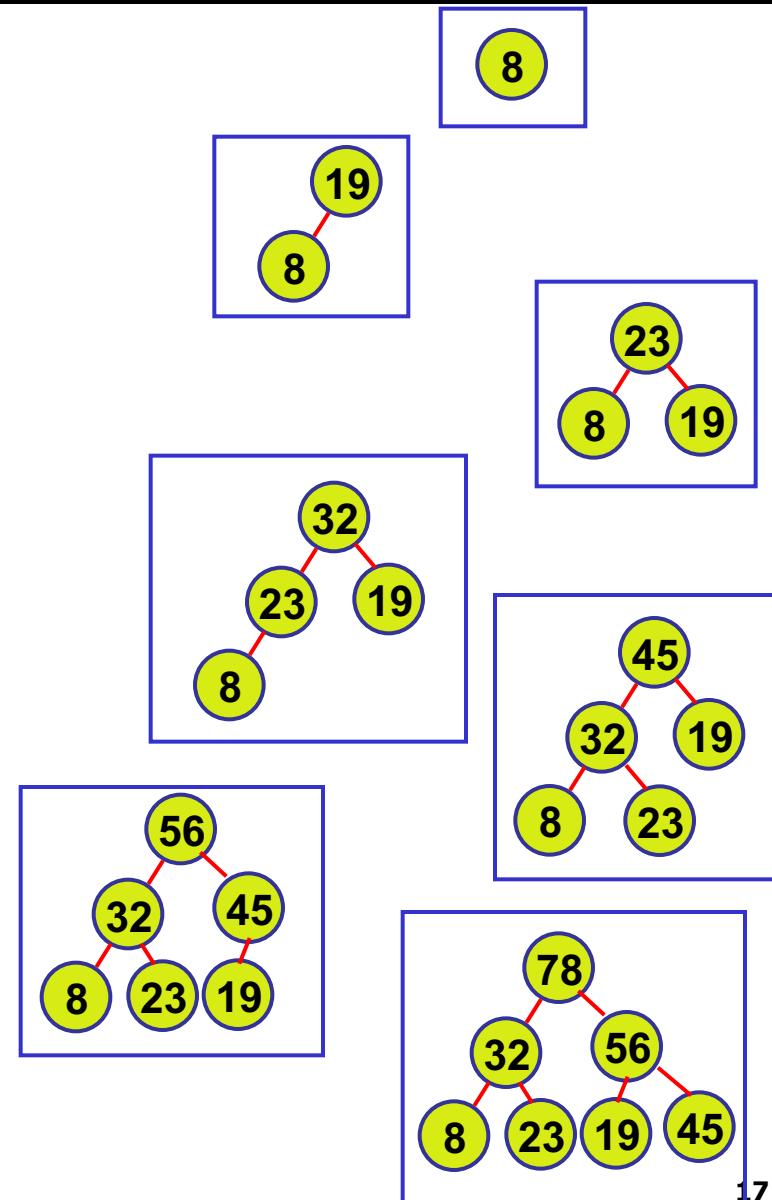
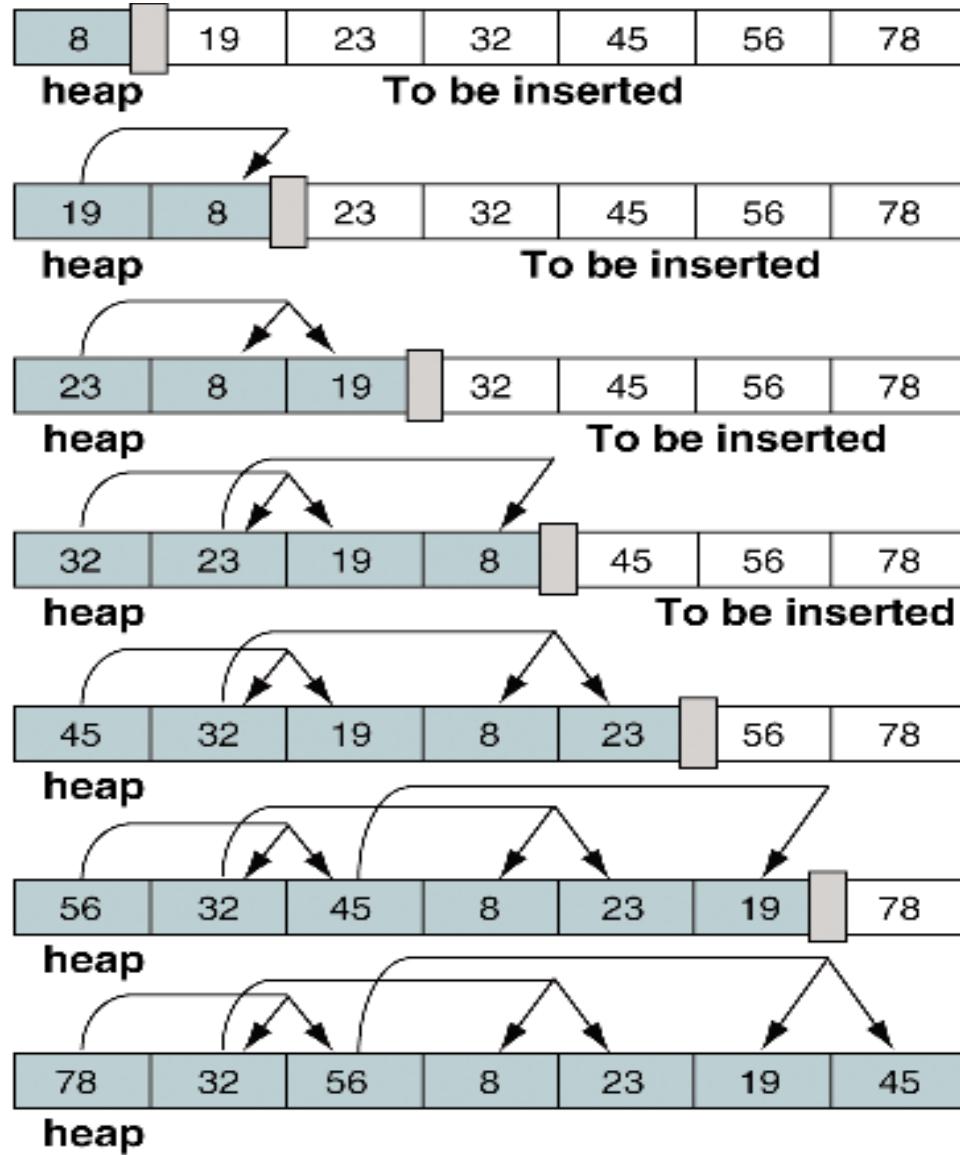
 1 reheapUp(heap, walker)

 2 increment walker

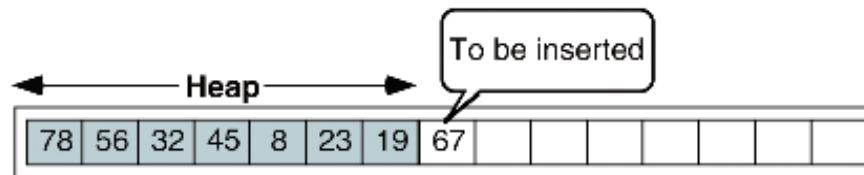
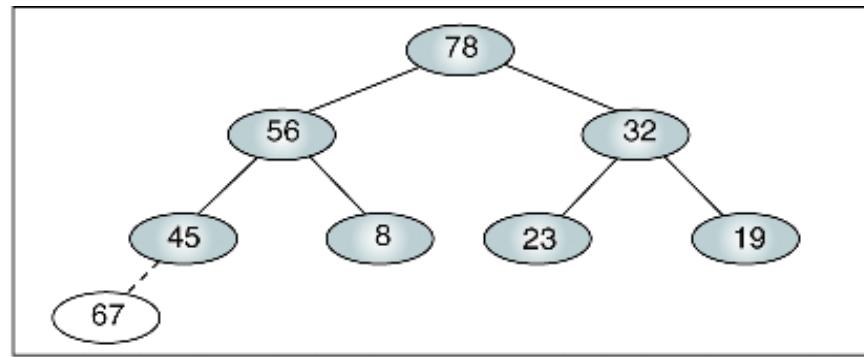
3 end loop

end buildHeap

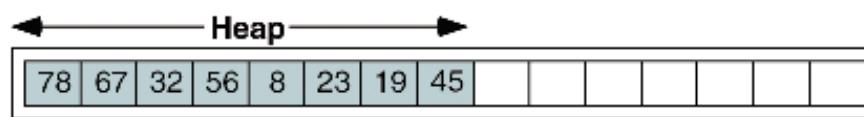
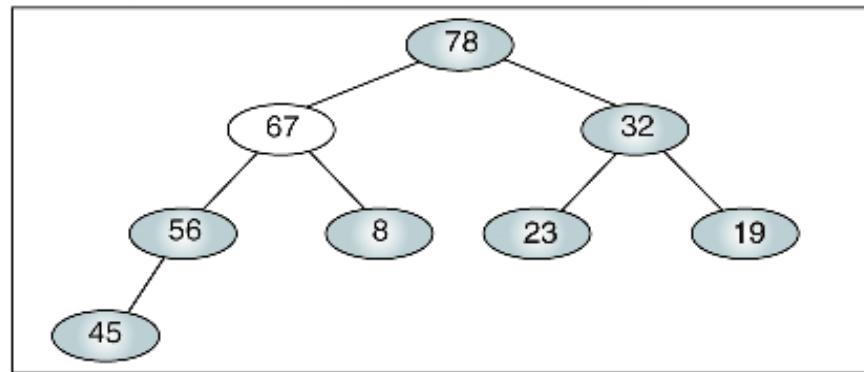
Building a Heap



Insert Node



(a) Before reheap up



(b) After reheap up

Insert Heap

```
Algorithm insertHeap (heap, last, data)
```

Inserts data into heap.

Pre heap is a valid heap structure

 last is reference parameter to last node in heap

 data contains data to be inserted

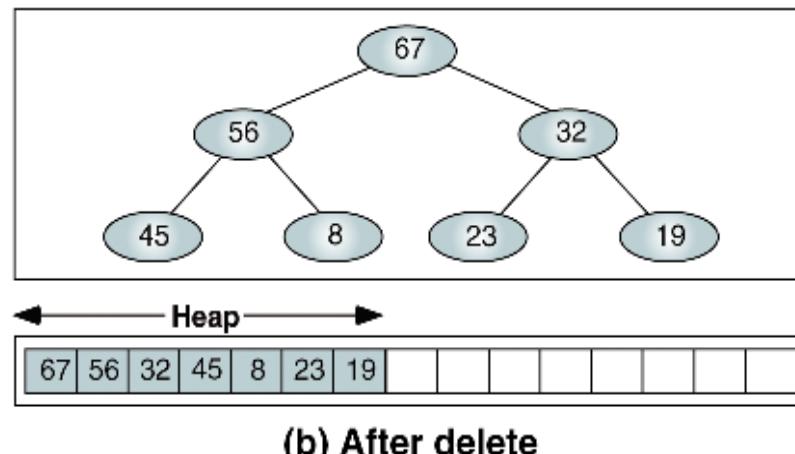
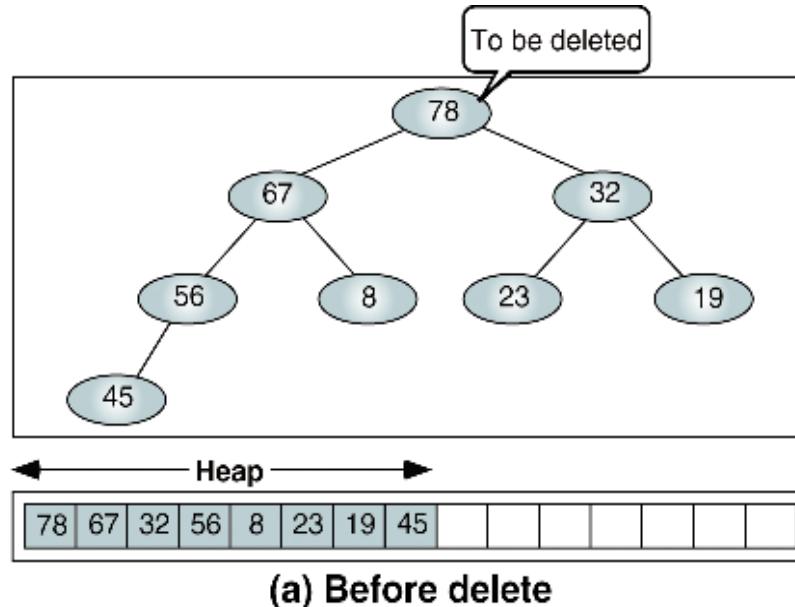
Post data have been inserted into heap

Return true if successful; false if array full

```
1 if (heap full)
  1 return false
2 end if
3 increment last
4 move data to last node
5 reheapUp (heap, last)
6 return true
end insertHeap
```

Delete Heap Node

- The most common and meaningful logic is to delete the root



Delete Heap Node

```
Algorithm deleteHeap (heap, last, dataOut)
Deletes root of heap and passes data back to caller.

    Pre    heap is a valid heap structure
           last is reference parameter to last node in heap
           dataOut is reference parameter for output area

    Post   root deleted and heap rebuilt
           root data placed in dataOut
           Return true if successful; false if array empty

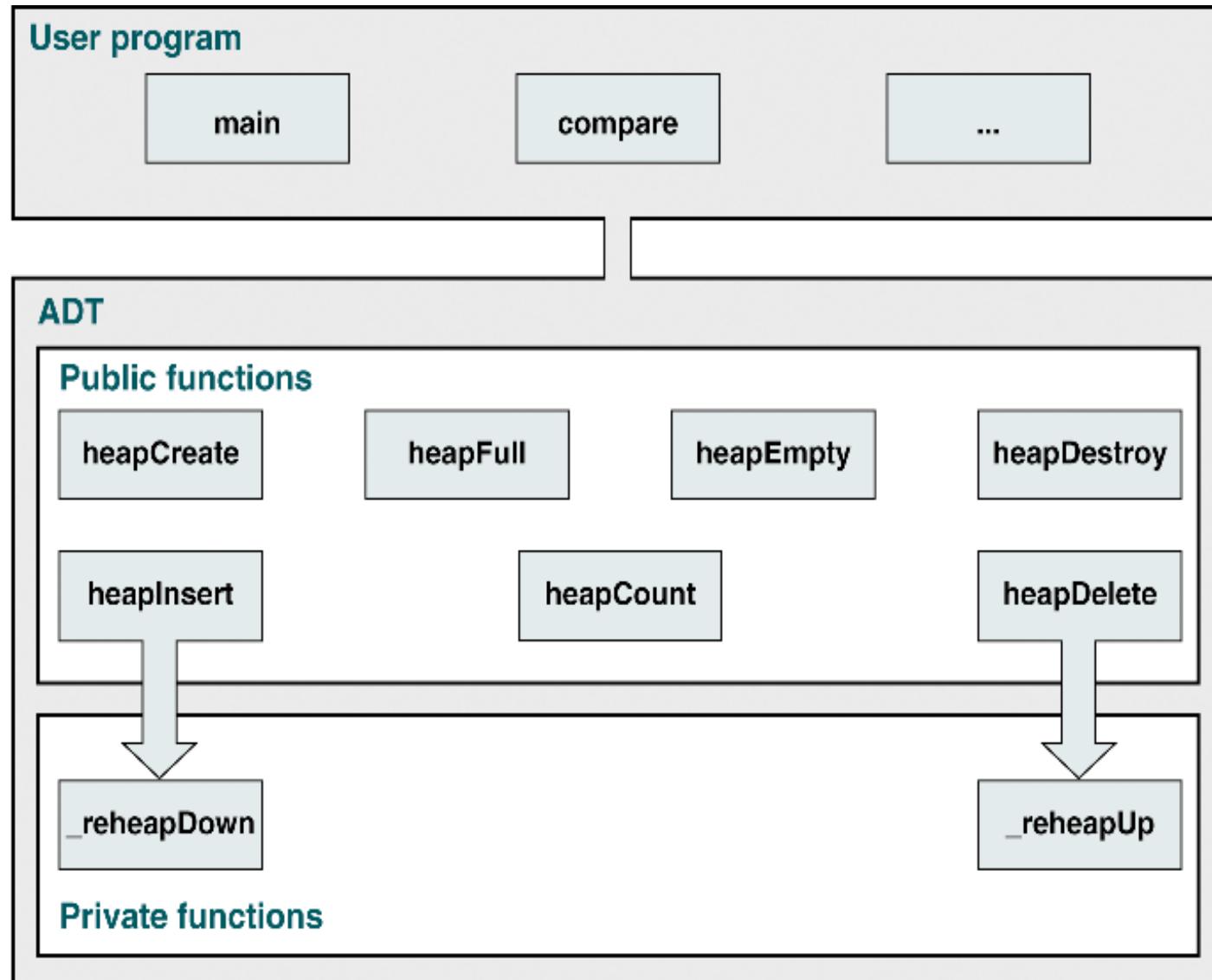
1  if (heap empty)
   1  return false
2  end if
3  set dataOut to root data
4  move last data to root
5  decrement last
6  reheapDown (heap, 0, last)
7  return true
end deleteHeap
```

9-3 Heap ADT

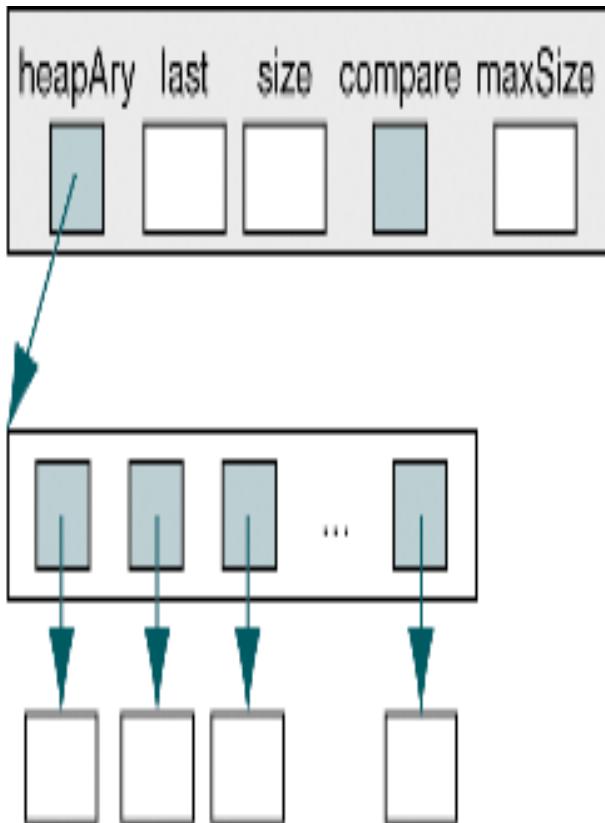
We begin with a discussion of the heap ADT design and then develop the C code for the five major functions developed in Section 9.2. Basic functions, such as heap full, are not provided.

- **Heap Structure**
- **Heap Algorithms**

Heap ADT Design



Heap ADT Structure



```
typedef struct
{
    void** heapAry;
    int    last;           []
    int    size;
    int    (*compare)(void* arg1, void* arg2);
    int    maxSize;
} HEAP;
```

Heap Declaration

```
5 #include <stdbool.h>
6
7 typedef struct
8 {
9     void** heapAry;
10    int      last;
11    int      size;
12    int      (*compare) (void* arg1, void* argu2);
13    int      maxSize;
14 } HEAP;
15
16 // Prototype Definitions
17 HEAP* heapCreate (int maxSize,
18                     int (*compare) (void* arg1, void* arg2));
19 bool  heapInsert   (HEAP* heap, void* dataPtr);
20 bool  heapDelete   (HEAP* heap, void** dataOutPtr);
21 int   heapCount   (HEAP* heap);
22 bool  heapFull    (HEAP* heap);
23 bool  heapEmpty   (HEAP* heap);
24 void  heapDestroy (HEAP* heap);
25
26 static void _reheapUp   (HEAP* heap, int childLoc);
27 static void _reheapDown (HEAP* heap, int root);
```

Create Heap Application interface

```
1  /* ===== heapCreate =====
2   Allocates memory for heap and returns address of
3   heap head structure.
4   Pre Nothing
5   Post heap created and address returned
6           if memory overflow, NULL returned
7 */
8 #include <math.h>
9
10 HEAP* heapCreate (int maxSize,
11                   int (*compare) (void* argu1, void* argu2))
12 {
13 // Local Definitions
14     HEAP* heap;
```

Create Heap Application interface (cont.)

```
16 // Statements
17 heap = (HEAP*)malloc(sizeof (HEAP));
18 if (!heap)
19     return NULL;
20
21 heap->last      = -1;
22 heap->compare = compare;
23
24 // Force heap size to power of 2 -1
25 heap->maxSize =
26             (int) pow (2, ceil(log2(maxSize))) - 1;
27 heap->heapAry = (void*)
28                     calloc(heap->maxSize, sizeof(void*));
29 return heap;
30 } // createHeap
```

Insert Heap Application interface

```
1  /* ===== heapInsert =====
2   Inserts data into heap.
3     Pre    Heap is a valid heap structure
4             last is pointer to index for last element
5             data is data to be inserted
6     Post   data have been inserted into heap
7             Return true if successful; false if array full
8 */
9  bool heapInsert (HEAP* heap, void* dataPtr)
10 {
11 // Statements
12     if (heap->size == 0)                      // Heap empty
13     {
14         heap->size = 1;
15         heap->last = 0;
16         heap->heapAry[heap->last] = dataPtr;
17         return true;
18     } // if
19     if (heap->last == heap->maxSize - 1)
20         return false;
21     ++(heap->last);
22     ++(heap->size);
23     heap->heapAry[heap->last] = dataPtr;
24     _reheapUp (heap, heap->last); // Line 24
25     return true;
26 } // heapInsert
```

Insert ReHeap Up Function

```
1  /* ===== reheapUp =====
2   Reestablishes heap by moving data in child up to
3   correct location heap array.
4   Pre  heap is array containing an invalid heap
5           newNode is index to new data in heap
6   Post newNode inserted into heap
7 */
8 void _reheapUp (HEAP* heap, int childLoc)
9 {
10 // Local Definitions
11     int    parent;
12     void** heapAry;
13     void*  hold;
14 }
```

Insert ReHeap Up Function (cont.)

```
15 // Statements
16     // if not at root of heap -- index 0
17     if (childLoc)
18     {
19         heapAry = heap->heapAry;
20         parent = (childLoc - 1)/ 2;
21         if (heap->compare(heapAry[childLoc],
22                             heapAry[parent]) > 0)
23             // child is greater than parent -- swap
24             {
25                 hold          = heapAry[parent];
26                 heapAry[parent] = heapAry[childLoc];
27                 heapAry[childLoc] = hold;
28                 _reheapUp (heap, parent);
29             } // if heap[]
30     } // if newNode
31     return;
32 } // reheapUp
```

Delete Heap Application Interface

```
1  /* ===== heapDelete =====
2   Deletes root of heap and passes data back to caller.
3   Pre    heap is a valid heap structure
4           last is reference to last node in heap
5           dataOut is reference to output area
6   Post   last deleted and heap rebuilt
7           deleted data passed back to user
8           Return true if successful; false if array empty
9 */
10  bool heapDelete (HEAP* heap, void** dataOutPtr)
11  {
12  // Statements
13      if (heap->size == 0)
14          // heap empty
15          return false;
16      *dataOutPtr = heap->heapAry[0];
17      heap->heapAry[0]  = heap->heapAry[heap->last];
18      (heap->last)--;
19      (heap->size)--;
20      _reheapDown (heap, 0);
21      return true;
22 } // heapDelete
```

Internal Reheap Down Function

```
1  /* ===== reheapDown =====
2   Reestablishes heap by moving data in root down to its
3   correct location in the heap.
4   Pre  heap is array of data
5           root is root of heap or subheap
6           last is an index to last element in heap
7   Post heap has been restored
8 */
9  void _reheapDown (HEAP* heap, int root)
10 {
11 // Local Definitions
```

Internal Reheap Down Function (cont.)

```
12     void* hold;
13     void* leftData;
14     void* rightData;
15     int   largeLoc;
16     int   last;
17
18 // Statements
19 last = heap->last;
20 if ((root * 2 + 1) <= last)           // left subtree
21     // There is at least one child
22 {
```

Internal Reheap Down Function (cont.)

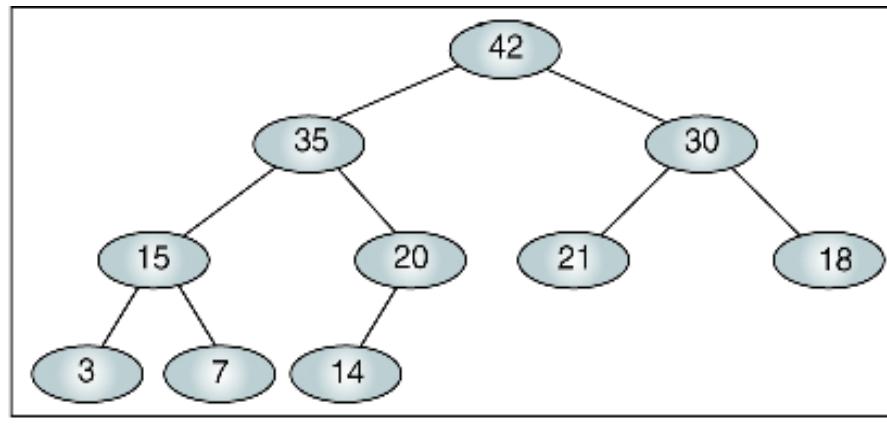
```
23     leftData    = heap->heapAry[root * 2 + 1];
24     if ((root * 2 + 2) <= last) // right subtree
25         rightData = heap->heapAry[root * 2 + 2];
26     else
27         rightData = NULL;
28
29     // Determine which child is larger
30     if (!rightData)
31         || heap->compare (leftData, rightData) > 0
32     {
33         largeLoc = root * 2 + 1;
34     } // if no right key or leftKey greater
35     else
36     {
37         largeLoc = root * 2 + 2;
38     } // else
39     // Test if root > larger subtree
40     if (heap->compare (heap->heapAry[root],
41                         heap->heapAry[largeLoc]) < 0)
42     {
43         // parent < children
44         hold = heap->heapAry[root];
45         heap->heapAry[root] =
46             heap->heapAry[largeLoc];
47         heap->heapAry[largeLoc] = hold;
48         _reheapDown (heap, largeLoc);
49     } // if root <
50 } // if root
51 return;
52 } // reheapDown
```

9-4 Heap Applications

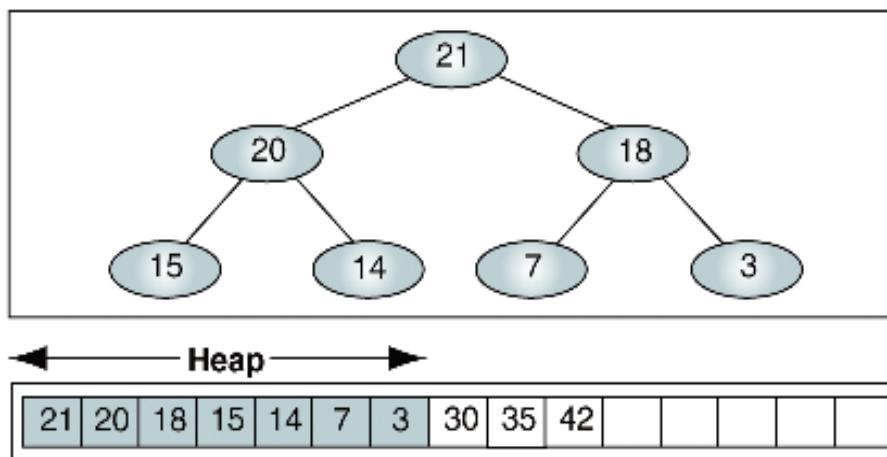
We discuss two of the three common heap applications-selection and priority queues. For selection applications, we develop a high-level algorithm. For priority queues, we develop the C code.

- Selection Algorithms
- Priority Queues
- Sorting (Chapter 12)

Heap Selection



(a) Original heap

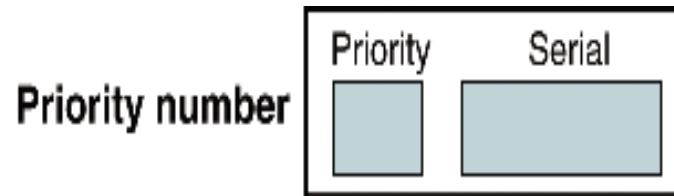


(b) After three deletions

Heap Selection

```
Algorithm selectK (heap, k, heapLast)
Select the k-th largest element from a list
    Pre    heap is an array implementation of a heap
           k is the ordinal of the element desired
           heapLast is reference parameter to last element
    Post   k-th largest value returned
1 if (k > heap size)
1 return false
2 end if
3 set origHeapSize to heapLast + 1
4 loop (k times)
    1 set tempData to root data
    2 deleteHeap (heap, heapLast, dataOut)
    3 move tempData to heapLast + 1
5 end loop
    Desired element is now at top of heap
6 move root data to holdOut
    Reconstruct heap
7 loop (while heapLast < origHeapSize)
    1 increment heapLast
    2 reheapUp (heap, heapLast)
8 end loop
9 return holdOut
end selectK
```

Priority Queue Priority Numbers

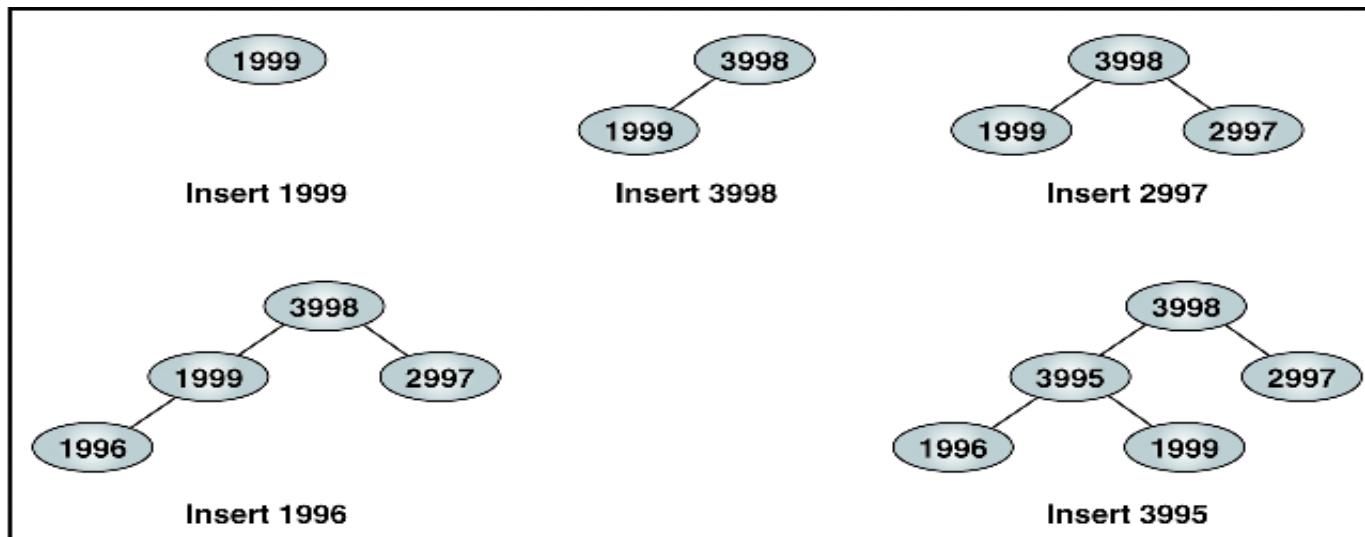


Priority	Serial	Priority	Serial	Priority	Serial
1	999	3	999	5	999
.
.
.
1	000	3	000	5	000
2	999	4	999		
.	.	.	.		
.	.	.	.		
.	.	.	.		
2	000	4	000		

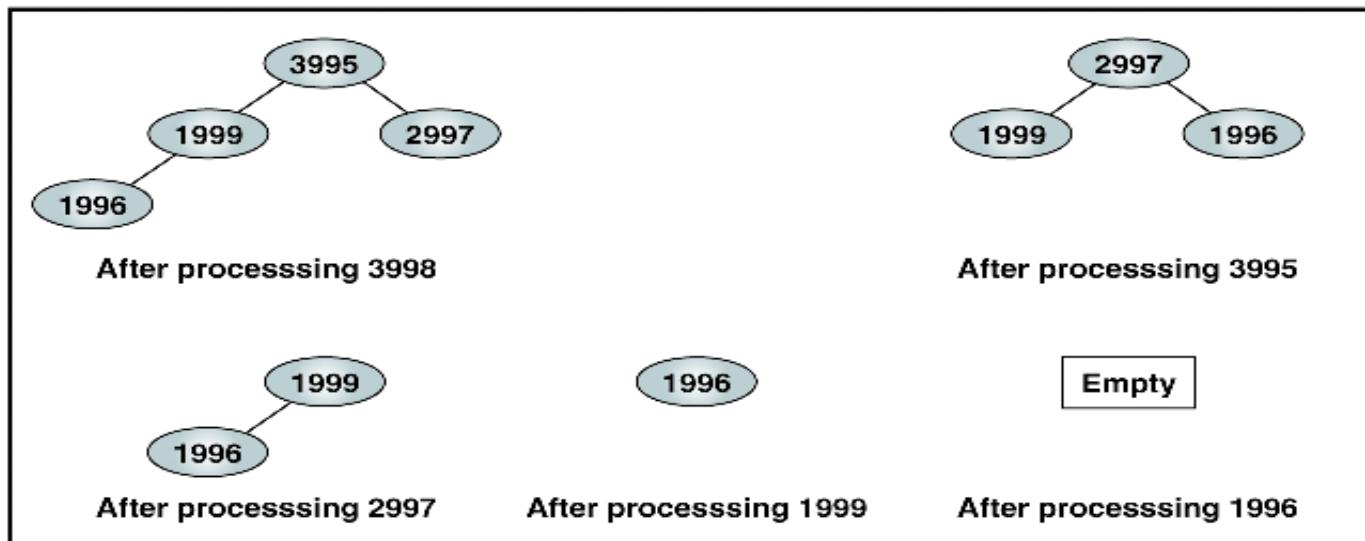
Priority Number Assignments

Arrival	Priority	Priority
1	low	1999 (1 & (1000 - 1))
2	high	3998 (3 & (1000 - 2))
3	medium	2997 (2 & (1000 - 3))
4	low	1996 (1 & (1000 - 4))
5	high	3995 (3 & (1000 - 5))

Priority Queue Example

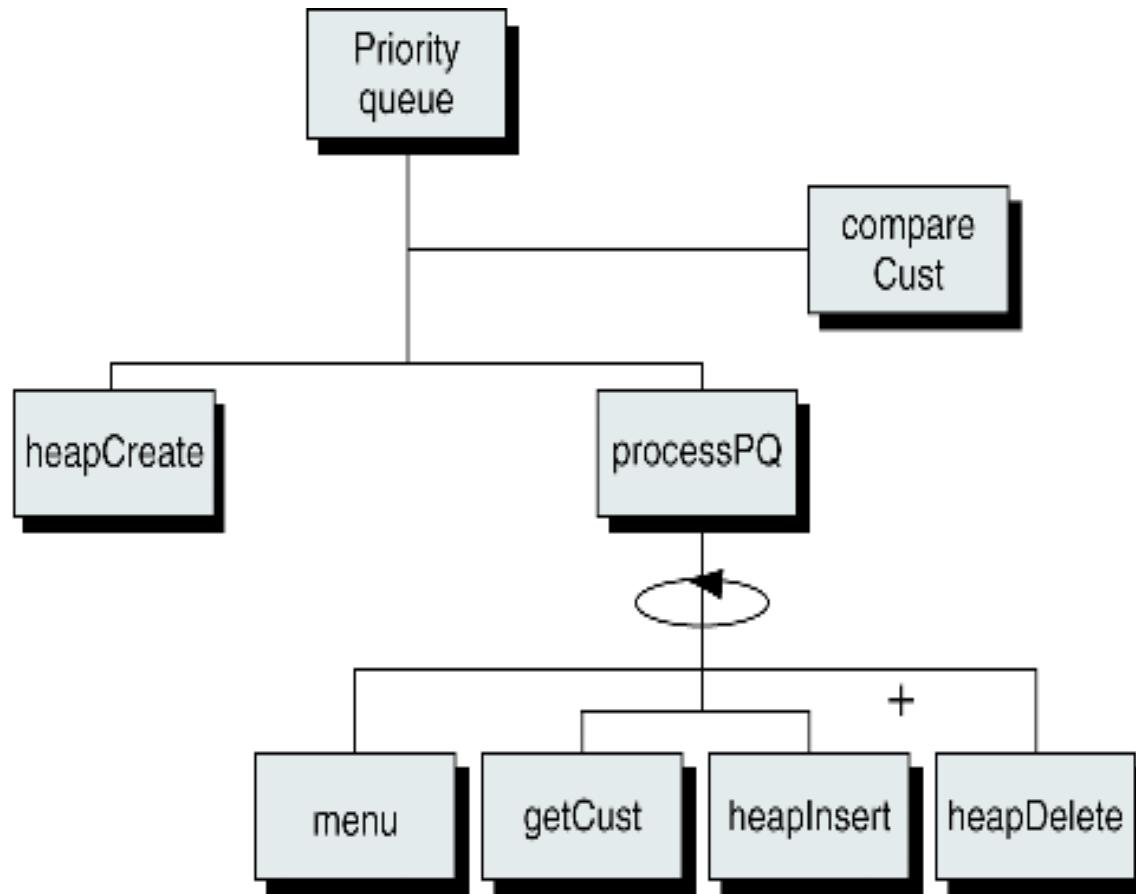


(a) Insert customers



(b) Process customers

Priority Queue Design



Priority Queue Implementation

```
1  /* Implement priority queue using heap.  
2   Written by:  
3   Date:  
4 */  
5  #include <stdio.h>  
6  #include <ctype.h>  
7  #include <stdbool.h>  
8  
9  #include "P9-heap.h"  
10  
11 // Constant Definitions  
12 const int maxQueue = 20;  
13  
14 // Structure Declarations  
15 typedef struct  
16 {  
17     int id;  
18     int priority;  
19     int serial;  
20 } CUST;
```

Priority Queue Implementation (cont.)

```
21
22 // Prototype Declarations
23 int compareCust (void* cust1, void* cust2);
24 void processPQ (HEAP* heap);
25 char menu (void);
26 CUST* getCust (void);
27
28 int main (void)
29 {
30 // Local Definitions
31     HEAP* prQueue;
32
33 // Statements
34     printf("Begin Priority Queue Demonstration\n");
35
36     prQueue = heapCreate(maxQueue, compareCust);
37     processPQ (prQueue);
38
39     printf("End Priority Queue Demonstration\n");
40     return 0;
41 } // main
```

Priority Queue Implementation (cont.)

```
42  /* ===== compare =====
43   Compare priority of two customers to determine
44   who has higher priority.
45   Pre Given two customer structures
46   Post if cust1 > cust2 return +1
47       if cust1 == cust2 return 0
48       if cust1 < cust2 return -1
49 */
50 int compareCust (void* cust1, void* cust2)
51 {
52 // Local Definitions
53     CUST c1;
54     CUST c2;
55
56 // Statements
57     c1 = *(CUST*)cust1;
58     c2 = *(CUST*)cust2;
59
60     if (c1.serial < c2.serial)
61         return -1;
62     else if (c1.serial == c2.serial)
63         return 0;
64     return +1;
65 } // compareCust
66
67
```

Priority Queue Implementation (cont.)

```
68 /* ===== processPQ =====
69 Compare priority of two customers to determine
70 who has higher priority.
71     Pre Given two customer structures
72     Post if cust1 > cust2 return +1
73             if cust1 == cust2 return 0
74             if cust1 < cust2 return -1
75 */
76 void processPQ (HEAP* prQueue)
77 {
78 // Local Definitions
79     CUST* cust;
80     bool result;
81     char option;
82     int numCusts = 0;
83 }
```

Priority Queue Implementation (cont.)

```
84 // Statements
85     do
86     {
87         option = menu ();
88         switch (option)
89         {
90             case 'e':
91                 cust = getCust ();
92                 numCusts++;
93                 cust->serial =
94                     cust->priority * 1000 + (1000 - numCusts);
95                 result = heapInsert (prQueue, cust);
96                 if (!result)
97                     printf("Error inserting into heap\n"),
98                     exit (101);
99                 break;
100            case 'd':
101                result = heapDelete (prQueue, (void**) &cust);
102                if (!result)
103                    printf("Error: customer not found\n");
```

Priority Queue Implementation (cont.)

```
104         else
105             {
106                 printf("Customer %4d deleted\n",
107                         cust->id);
108                 numCusts--;
109             } // else
110         } // switch
111     } while (option != 'q');
112     return;
113 } // processPQ
114
```

Priority Queue Implementation (cont.)

```
115  /* ===== menu ===== */
116  Display menu and get action.
117  Pre nothing
118  Post action read and validated
119 */
120 char menu (void)
121 {
122 // Local Declarations
123     char option;
124     bool valid;
125
126 // Statements
127     printf( "\n===== Menu =====\n" );
128     printf( " e : Enter Customer Flight \n" );
129     printf( " d : Delete Customer Flight \n" );
130     printf( " q : Quit. \n" );
131     printf( "===== \n" );
132     printf( "Please enter your choice: " );
```

Priority Queue Implementation (cont.)

```
133
134     do
135         {
136             scanf(" %c", &option);
137             option = tolower (option);
138             switch (option)
139                 {
140                     case 'e':
141                     case 'd':
142                     case 'q': valid = true;
143                         break;
144
145                     default: printf("Invalid choice. Re-Enter: ");
146                         valid = false;
147                         break;
148                 } // switch
149             } while (!valid);
150         return option;
151     } // menu
```

Priority Queue Implementation (cont.)

```
152
153 /* ===== getCustomer =====
154     Reads customer data from keyboard.
155     Pre  nothing
156     Post data read and returned in structure
157 */
158 CUST* getCustomer (void)
159 {
160 // Local Definitions
161     CUST* cust;
```

Priority Queue Implementation (cont.)

```
162
163 // Statements
164     cust = (CUST*)malloc(sizeof (CUST));
165     if (!cust)
166         printf("Memory overflow in getCust\n"),
167         exit (200);
168
169     printf("Enter customer id:      ");
170     scanf ("%d", &cust->id);
171     printf("Enter customer priority: ");
172     scanf ("%d", &cust->priority);
173     return cust;
174 } // getCust
```