目录

Heaps

The segment of memory that can be used by a C programmer is referred to as the ***heap***

- more accurately, it is called ***heap memory***
- a programmer can request heap memory by using *malloc()*
    - for example, when creating a node for a linked list
- the memory is 'returned' to the heap by using *free()*

That has nothing to do with the ***heap data structure***.

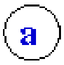
A heap data structure is a high-level, two-dimensional data structure

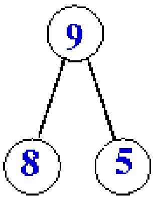
- it is 'higher-level' than an array, or a linked list ...
    - in the sense that a heap data structure is built on an array, or linked list
- it is two-dimensional because it forms a tree of data
    - a binary tree (at most 2 children per node)

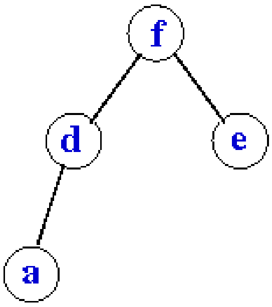
A heap:

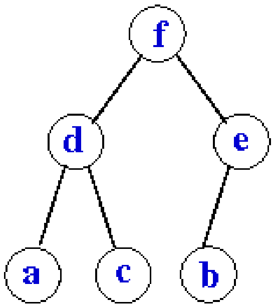
- has a root node
- the root node can have 0, 1 or 2 'child' nodes
- in turn, each child node can have 0, 1 or 2 child nodes

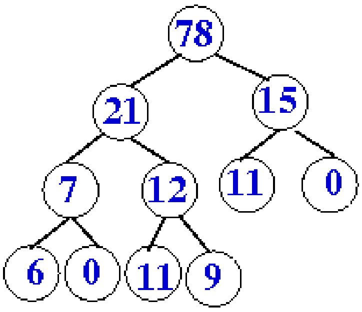
Examples of heaps:









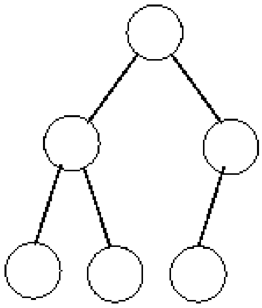


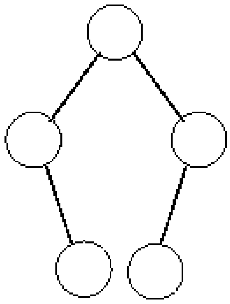
Types of heaps:

- **Max-heaps**
    - the element with the largest key is at the root
    - the children of a node have keys smaller than the parent
- **Min-heaps**
    - the element with the smallest key is at the root
    - the children of a node have keys larger than the parent

A max-heap (analogously for min-heap) satisfies 2 properties:

1. ***Heap-Order Property*** (sometimes called HOP)
    - for root node with key *k*
        - the nodes in each subtree has key ≤ *k*
    - the root of each subtree recursively down the tree satisfies the same property
2. ***Complete-Tree Property*** (sometimes called CTP)
    - nodes in a given level are filled in from left to right with no breaks
    - every level is filled in before adding a node to the next level
        - technically should be called the ***Almost-Complete Tree Property*** ...
            - because 'complete' means exactly 2 children per non-leaf node





A heap data structure is ***partially ordered***

- more order than 'unordered', less ordered than 'ordered'
- useful in applications that don't need fully ordered data
  - the application may want the maximum or minimum element (only)
    - sorting a whole list just to find the maximum is very inefficient
- heaps often used to implement ***priority queues***, used in shortest-path algorithms

Size of a heap:

- because of the Complete-Tree Property,
  - at level 0 there is 2^0 nodes (i.e. the root node)
  - at level 1 there is either 1 or 2^1 nodes (i.e. the children of the root node)
  - at level 2 there are between 1 and 2^2 nodes (i.e. 'grand-children' of the root node)
  - at level 3 there are between 1 and 2^3 nodes
  - ...
  - at level n there are between 1 and 2^n nodes

If there are a total of N nodes then the maximum value of n is O(log(N))

- because approximately $2^n = N$ hence $n = \log(N)$

This means that:

```
the maximum height of a heap of size N is O(log(N))
```

This means that:

```
the maximum path length in a heap of size N is O(log(N))
```

Heap Implementations

An array is the most efficient way of implementing a heap data structure

Ironically, a heap data structure implemented on a linked list ...

- would be using heap memory, but this is rarely done in practice

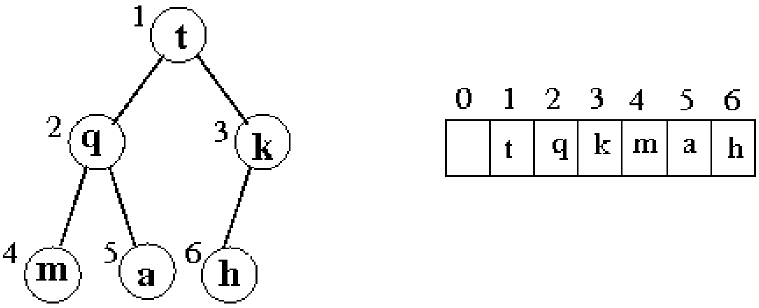
Why?

- because of the Complete Tree Property, CTP

We always place the root of the heap at index 1

- given any node at index **i**:

- its left child is located at **2i**
- its right child is located at **2i+1**
- its parent is located at **i/2** (use integer division)



- Here:
  - **t** (**i**=1) has children **q** (**2*i**=2*1) and **k** (**2*i+1**=2*1+1)
  - **q** (2) has children **m** (2*2) and **a** (2*2+1)
  - **k** (3) has one child **h** (2*3)
  - parent of **q** (**i**=2) is **t** (**i/2**=2/2)
  - parent of **k** (3) is **t** (3/2)
  - parent of **m** (4) is **q** (4/2)
  - parent of **a** (5) is **q** (5/2)
  - parent of **h** (6) is **k** (6/2)

Operation on heaps

Get the maximum element

The maximum element is always the root element, which is always at location 1

- a trivial, constant-time operation
  - return element at location 1

Note, we are just 'getting' the maximum element

- we are not changing the heap in any way

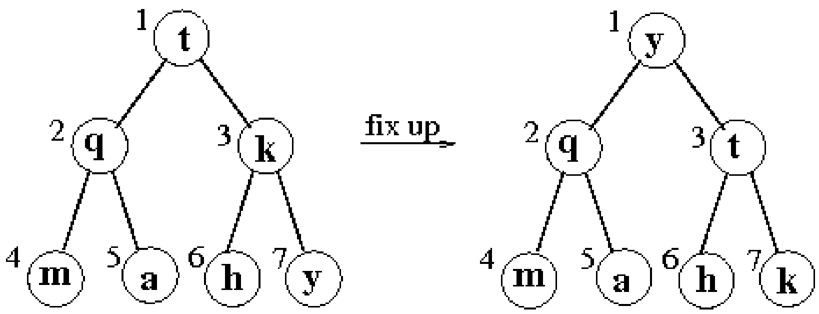
Insert an element

Inserting an element in a heap is a 2-stage process:

1. **put** the new element at bottom-most, rightmost position (so *Complete-Tree Property* is maintained).
   - in other words, add it to the end of the array
2. **fixUp** (i.e. possibly swap) values along single path back to root to restore the *Heap-Order Property*

For example, assume we want to add the key 'y' to the heap

1. **put** the new key at the bottom right position (i.e. at location 7)
   - this makes the heap incorrect *why?*
2. we do a **fix up** to re-establish order
   1. first swap **k** and **y**
   2. next swap **t** and **y**



fixUp

在最后添加元素，与上面的父节点比较，如果太大就交换，直到不大了为止

- is also called **heapify up**
- means simply to move a key to its correct place upwards in the heap
- has complexity *O(log(N))* because that is the height of the heap

Let us implement the heap data structure in an array called *heap[]*

- the root of the heap is at *heap[1]*
- assume we have a new node ...
  - we put it at the end of the array, at index **child** say
    - we know the parent of the new node is at index **child/2**
  - then we **fixUp** the tree order by calling the function:

只有当满足两个条件的时候，才会逐渐向上移动，否则直接停止就行，并不是一定要比较到最上面

切换行号显示

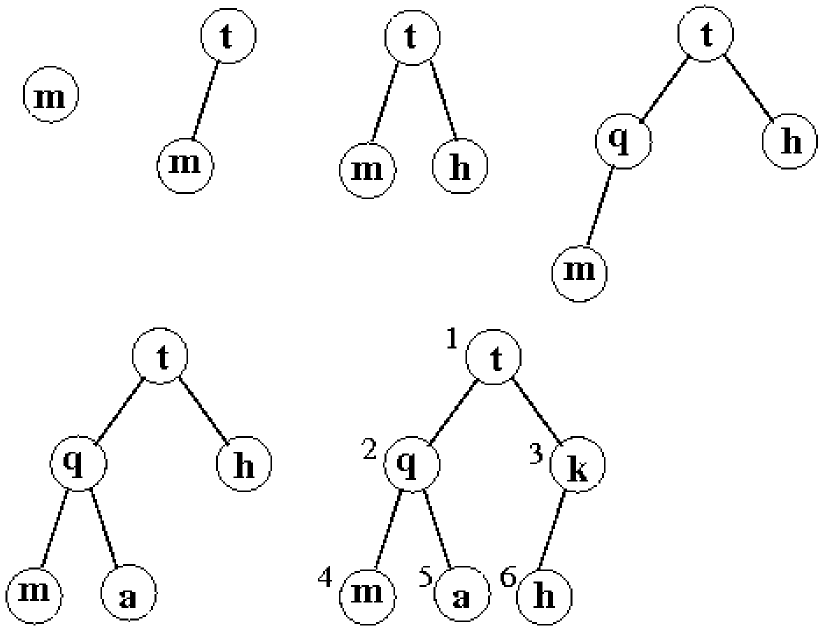
```
// fix up the heap for the new element at index child
void fixUp(int *heap, int child) {
   while (child>1 && heap[child/2]<heap[child]) {
      int swap = heap[child];      // if parent < child ...
      heap[child] = heap[child/2]; // swap them ...
      heap[child/2] = swap;        // and then ...
      child = child/2;             // become the parent
   }
   return;
}
```

Example

We'll make a heap by inserting the keys **m**, **t**, **h**, **q**, **a** and **k**

1. Start with an empty heap. Make the first node **m** the root node (see diagram below)
2. Add **t**
   - Completeness: tree is a root node with left child
   - Heap-Order violated?
     - yes, need to *fix up* (which will swap **t** and **m**)
3. Add **h**
   - Completeness: add to next available position
   - Head-Order violated?
     - no
4. Add **q**
   - Completeness: add to next available position
   - Heap-Order violated?
     - yes, need to *fix up* (which will swap **q** and **m**)
5. Add **a**
   - Completeness: add to next available position

   - Head-Order violated?
      - no
6. Add **k**
   - Completeness: add to next available position
   - Head-Order violated?
      - yes, need to *fix up* (which will swap **k** and **h**)



Items inserted in order: m t h q a k

Delete the maximum element

This operation if often referred to as:

- **delMax** if it is a max-heap or
- **delMin** if it is a min-heap

If we were deleting a leaf node, it is 'easy' in general

- the parent node simply deletes the link to the child

Deleting the root node, which usually has children, is difficult

- what do you do with the children?

To delete the maximum element, we must delete the root node of the heap.

The function carries out 4 steps:

1. generally the function returns the node that is deleted
   - so **save the root node** into a temporary variable
   - put the last element in the heap into the root position (index 1)
      - which is the bottom-most, rightmost element in the heap
      - which is the last element in the array
         - the array is of size *n*, so location of last element is *n*
   - the heap order property (HOP) must now be broken
2. **delete the last element**
   - this can be done by just changing the size of the heap
   - deleting the last element means the CTP is maintained

3. **fixDown from the root**
    - this ensures the HOP is restored
4. **return the saved root node**
    - this was saved in the first step and is now returned

Code to implement this is quite simple:

切换行号显示

```
int delMaxHeap(int *heap, int len) {
   int retval = heap[1];     // remember the root
   heap[1] = heap[len];      // replace the root
   fixDown(heap, 1, len-1);  // fixDown, heap now has length 'len-1'
   return retval;
}
```

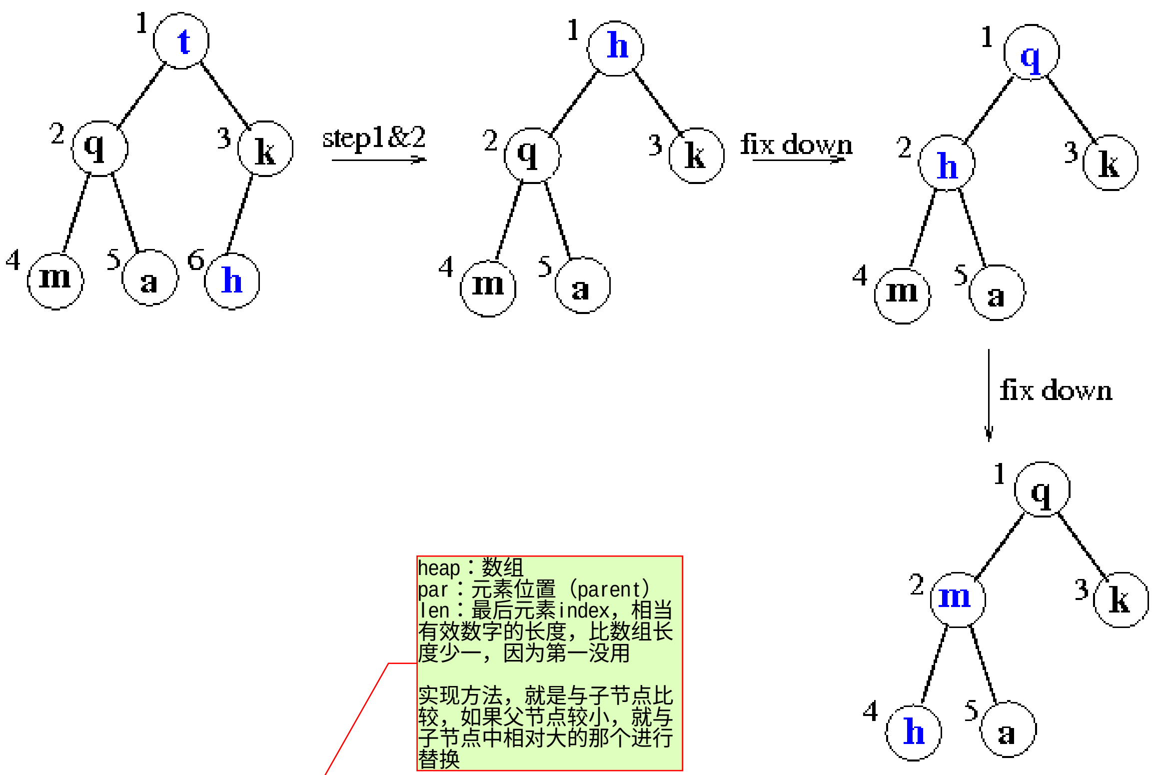
fixDown

- also called **heapify down**
- if a node is smaller than either of its children, swap with the largest child
    - means simply to move a key to its correct place downwards in the heap
- has complexity *O(log(N))* because that is the height of the heap

Example

When we delete the root element in the next example, we:

- over-write the root with the bottom-right element
- and then do a fixDown, consisting of two swaps



heap：数组
par：元素位置（parent）
len：最后元素index，相当有效数字的长度，比数组长度少一，因为第一没用

实现方法，就是与子节点比较，如果父节点较小，就与子节点中相对大的那个进行替换

切换行号显示

```
// force value at a[par] into correct position
void fixDown(int *heap, int par, int len) {
```

```
    int finished = 0;
    while (2*par<=len && !finished) {// as long as you have a child
       int child = 2*par;           // the first child is here
       if (child<len && heap[child]<heap[child+1]) {
          child++;                  // choose larger of two children
       }
       if (heap[par]<heap[child]) {// compare parent with largest child
          int swap = heap[child];  // if parent<child, do a swap
          heap[child] = heap[par];
          heap[par] = swap;
          par = child;              // ... and become this child
       }
       else {                       // if parent>child we are finished
          finished = 1;             //
       }
    }
    return;
}
```

Why do we need to swap with the largest child in a fixDown()?

- consider the following case
  - **if** we were to swap with the smallest of the children when we *fixDown()* then we get

```
        e                a               c
       / \   step       / \   swap      / \
      d   c  ====>     d   c  ====>    d   a
     / \     1&2      /       a&c     /
    b   a            b               b
```

  - the resulting tree is not a heap
    - *c* is larger than *a* but it is smaller than *d*
  - what we should have done is swap with the largest child

```
        e               a               d               d
       / \   step      / \   swap      / \   swap      / \
      d   c  ====>    d   c  ====>    a   c  ====>    b   c
     / \     1&2     /       a&d     /       a&b     /
    b   a           b               b               a
```

    - because the largest child becomes the parent, it is guaranteed to not affect the other path
      - simply because the largest child by definition is larger than the other child

Animation of heap insertion/deletion

Somewhat slow 10 minute summary of insertion/deletion

Comparing heaps with 'normal' arrays

Suppose that we are interested in quickly doing operations:

- inserting elements into an array and
- deleting the maximum element (or minimum element) in an array

How does having elements in a heap (array) compare with having them in 'normal' arrays?

a. If the array is unordered:
    - we can insert items to the array in *O(1)* time (just add it to the end of the array)
    - to get the maximum item, we must do a linear search: this is **O(n)**
b. If the array is ordered:
    - we can insert items to the array in **O(n)** time if we use linear search (need to find the place to insert the item)
        - (or *O(log(n))* time if we use binary search)
    - to get the maximum item, we just return the last item: *O(1)* time
c. If the array has heap-order:
    - we can insert items in **O(log(n))** time
    - we can get the maximum item in **O(log(n))** time

So both operations are fast with heaps.

With ordered and unordered arrays, then you will suffer *O(n)* performance (which is very slow) in one of the operations (assuming you do not use binary search)

Converting an arbitrary array into a heap

If you already have an array of data, you can convert that into a max-heap (or min-heap) *in-place*, using 2 methods:

1. use *fixUp()* or
2. use *fixDown()*

All that these functions do is rearrange the data so that the elements satisfy HOP.

Example: consider the character array *heap[] = "roses"*

- represented as a heap we get

    ```
         r
       o   s
      e s
    ```

    which clearly is not a heap. There are 2 methods to make it a heap.
- Method 1: use **fixUp()**
    - we build the heap by inserting them one at a time
        - i.e. we **fixUp from 1 to n**

        ```
            r     r     r      s      s       s        s
                 o     o s    o r    o r     o r      s r
                                    e       e s      e o
                        s<->r                  s<->o
        ```

        - the array contains *ssreo*, which satifies the HOP
        - actually, *fixUp()* of location 1 does nothing because ... why?
        - so we could have done **fixUp()** from just 2 to *n*
- Method 2: use **fixDown()**
    - we consider each character in reverse order, and 'insert' into a heap using *fixDown()*
        - i.e. we **fixDown() from n to 1**

        ```
              r         r        s
        ```

```
     o s       s s      r s
    e s       e o      e o
        s<->o    s<->r
```

- the elements in the array *srseo* satify HOP
- actually, *fixDown()* from *n* up to *n/2+1* does nothing because ... why?
- so we could have done **fixDown()** from just *n/2* to 1

Notice that the heap arrays are different, but both are correct.

Do not get the direction wrong!

- *fixUp()* from *n* to *1* does not work!!
  - example:

```
      2   swap      6
    6   1 ====>   2   1
   4      6&2    4
```

- which is not a heap!
- *what went wrong?*
  - the functions *fixUp()* and *fixDown()* assume that we have heap order before elements are added
    - *fixUp()* of element *4* did nothing but it is on a path that fails HOP

Heaps (2019-07-09 09:09:34由AlbertNymeyer编辑)