Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Priority Queue using Heap

Dr. Shylaja S S
Ms. Kusuma K V

Priority Queue using Heap

Ascending Heap: Root has the lowest element. Each node's data is greater than or equal to its parent's data. It is also called **min heap**.

Descending Heap: Root has the highest element. Each node's data is lesser than or equal to its parent's data. It is also called **max heap**.

Priority Queue is a Data Structure in which intrinsic ordering of the elements does determine the results of its basic operations.

Ascending Priority Queue: is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed. If *apq* is an ascending priority queue, the operation *pqinsert(apq,x)* inserts element x into apq and *pqmindelete(apq)* removes the minimum element from *apq* and returns its value.

Descending Priority Queue: is a collection of items into which items can be inserted arbitrarily and from which only the largest item can be removed. If *dpq* is a descending priority queue, the operation *pqinsert(dpq,x)* inserts element x into *dpq* and *pqmaxdelete(dpq)* removes the maximum element from *dpq* and returns its value.

The operation *empty(pq)* applies to both types of priority queue and determines whether a priority queue is empty. *pqmindelete* or *pqmaxdelete* can only be applied to a non empty priority queue.

Once *pqmindelete* has been applied to retrieve the smallest element of an ascending priority queue, it can be applied again to retrieve the next smallest element, and so on. Thus the operation successively retrieves elements of a priority queue in ascending order (However, if a small element is inserted after several deletions, the next retrieval will return that smallest element, which may be smaller than a previously retrieved element). Similarly, *pqmaxdelete* retrieves elements of a descending priority queue in descending order. This explains the designation of a priority queue as either ascending or descending.

The elements of a priority queue need not be numbers or characters that can be compared directly. They may be complex structures that are ordered on one or several fields. For example, telephone-book listings consist of names, addresses, and phone numbers and are ordered by name.

Sometimes the field on which the elements of a priority queue are ordered is not even part of the elements themselves; it may be a special, external value used specifically for the purpose of ordering the priority queue. For example, a stack may be viewed as a descending priority queue whose elements are ordered by time of insertion. Te element that was inserted last has the greatest insertion-time value and is the only item that can be retrieved. A queue may similarly be viewed as an ascending priority queue whose elements are ordered by time of insertion. In both cases the time of insertion is not part of the elements themselves but is used to order the priority queue.

Heap as a Priority queue:

A heap allows a very efficient implementation of a priority queue. Now we will look into implementation of a descending priority queue using a descending heap. Let *dpq* be an array that implicitly represents a descending heap of size k. Because the priority queue is contained in array elements 0 to k-1, we add k as a parameter of insertion and deletion operations. Then the operation *pqinsert(dpq, k, elt)* can be implemented by simply inserting elt into its proper position in the descending list formed by the path from the root of the heap (dpq[0]) to the leaf dpq[k]. Once *pqinsert(dpq, k, elt)* has been executed, dpq becomes a heap of size k+1.

The insertion is done by traversing the path from the empty position k to position 0 (root), seeking the first element greater than or equal to *elt*. When that element is found, elt is inserted immediately preceding it in the path (i.e., *elt* is inserted as its child). As each element less than *elt* is passed during the traversal, it is shifted down one level in the tree to make room for *elt* (This shifting is necessary because we are using the sequential representation rather than a linked representation of the tree. A new element cannot be inserted between two existing elements without shifting some existing elements).

This heap insertion operation is also called the *siftup* operation because *elt* sifts its way up the tree. The following algorithm implements ***pqinsert(dpq, k, elt)***. **Algorithm for siftup**

```
c = k;
p = (c-1)/2;                    //p is parent of c
while(c>0 && dpq[p]<elt) {
  dpq[c]=dpq[p];
  c=p;                          //advance up the tree
  p=(c-1)/2;
}
dpq[c]=elt;
```

To implement *pqmaxdelete(dpq,k),* we note that the maximum element is always at the root of a k-element descending heap. When that element is deleted, the remaining k-1 elements in positions 1 through k-1 must be redistributed into positions 0 through k-2 so that the resulting array segment from dpq[0] through dpq[k-2] remains a descending heap. Let adjustheap(root,k) be the operation of rearranging the elements dpq[root+1] through dpq[k] into dpq[root] through dpq[k-1] so that subtree(root, k-1) forms a descending heap. Then ***pqmaxdelete(dpq,k)*** for a k-element descending heap can be implemented as:

```
p = dpq[0];
adjustheap(0,k-1);
return(p);
```

In a descending heap, not only is the root element the largest element in the tree, but an element in any position p must be the largest in subtree(p,k). Now subtree(p,k) consists of three groups of elements: its root, dpq[p]; its left subtree, subtree(2*p+1, k); and its right subtree, subtree(2*p+2, k). dpq[2*p+1], the left child of the root , is the largest element of the left subtree, and dpq[2*p+2], the right son of the root, is the largest element of the right subtree. When the root dpq[p] is deleted, the larger of these two children must move up to take its place as te new largest element of subtree(p,k). Then the subtree rooted at position of the larger element moved up must be readjusted in turn.

Algorithm largechild(p,m)

```
c = 2*p+1;
if(c+1 <= m && x[c] < x[c+1])
  c=c+1;
if(c > m)
  return -1;
else
  return (c);
```

Then, *adjustheap(root,k)* may be implemented recursively as:

Algorithm adjustheap(root,k) **//recursive**

```
p = root;
c = largechild(p,k-1);
if(c >= 0 && dpq[k] < dpq[c]){
  dpq[p] = dpq[c];
  adjustheap(c,k);
}
else
  dpq[p] = dpq[k];
```

adjustheap(root,k) may be implemented iteratively as:

```
p = root;
kvalue = dpq[k];
c = largechild(p,k-1);
while(c >= 0 && kvalue < dpq[c]){
    dpq[p] = dpq[c];
    p = c;
    c = largechild(p,k-1);
}
dpq[p] = kvalue;
```

//implementation of the priority queue using heap

```
#include<stdio.h>
#define MAX 50
typedef struct priq
{
 int pq[MAX];
 int n;
}PQ;
```

```
void init(PQ *pt)
{
 pt->n=0;
}
void disp(PQ *pt)
{
 int i;
 for(i=0;i<pt->n;i++)
  printf("%d ",pt->pq[i]);
}
int enqueue(PQ *pt,int e)
{int p,c;
  if(pt->n==MAX-1) return 0;

    c=pt->n;
    p=(c-1)/2;
    while(c>0 && pt->pq[p]<e)
      {
       pt->pq[c]=pt->pq[p];
        c=p;
        p=(c-1)/2;
      }
    pt->pq[c]=e;
    pt->n=pt->n+1;
    return 1;
}
int dequeue(PQ *pt,int *ele)
{
  int p,c;
  *ele=pt->pq[0];
  int elt=pt->pq[pt->n-1];
  p=0;
  if(pt->n==1)
    c=-1;
  else c=1;
   if(pt->n>2 && pt->pq[2]>pt->pq[1])
    c=c+1;
  while(c>=0 && elt<pt->pq[c])
   {
     pt->pq[p]=pt->pq[c];
     p=c;
     c=2*p+1;
```

```
      if(c+1<pt->n-1 &&  pt->pq[c+1]>pt->pq[c])
        c=c+1;
      if(c>=pt->n-1) c=-1;
     }
    pt->pq[p]=elt;
    pt->n=pt->n-1;
    return 1;
 }

int main()
{
 PQ pobj;
int k,choice,ele;

 init(&pobj);

 do{
   printf("1. Enqueue 2 Dequeue 3 Display\n");
   printf("Enter the choice");
   scanf("%d",&choice);
   switch(choice)
   {
   case 1: printf("enter the information");
           scanf("%d",&ele);
           enqueue(&pobj,ele);
           break;
   case 2: k=dequeue(&pobj,&ele);
          if(!k) printf("empty");
          else
           printf("%d dequeues element",ele);
          break;
   case 3: disp(&pobj);
            break;
  }
 }while(choice<4);

return 0;
}
```