Contents

Chapter 1

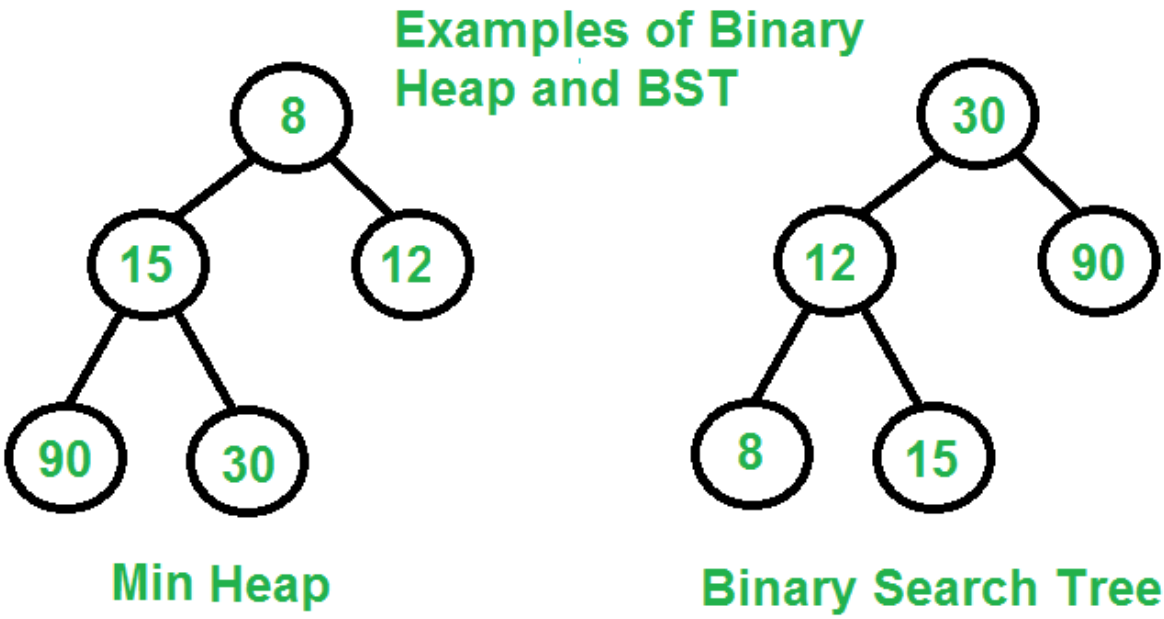
Why is Binary Heap Preferred over BST for Priority Queue?

A typical Priority Queue requires following operations to be efficient.

1. Get Top Priority Element (Get minimum or maximum)
2. Insert an element
3. Remove top priority element
4. Decrease Key

A Binary Heapsupports above operations with following time complexities:

1. O(1)
2. O(Logn)
3. O(Logn)
4. O(Logn)



A Self Balancing Binary Search Tree like AVL Tree, Red-Black Tree, etc can also support above operations with same time complexities.

1. Finding minimum and maximum are not naturally O(1), but can be easily implemented in O(1) by keeping an extra pointer to minimum or maximum and updating the pointer with insertion and deletion if required. With deletion we can update by finding inorder predecessor or successor.
2. Inserting an element is naturally O(Logn)
3. Removing maximum or minimum are also O(Logn)
4. Decrease key can be done in O(Logn) by doing a deletion followed by insertion. See this for details.

So why is Binary Heap Preferred for Priority Queue?

- Since Binary Heap is implemented using arrays, there is always better locality of reference and operations are more cache friendly.
- Although operations are of same time complexity, constants in Binary Search Tree are higher.
- We can build a Binary Heap in O(n) time. Self Balancing BSTs require O(nLogn) time to construct.
- Binary Heap doesn't require extra space for pointers.
- Binary Heap is easier to implement.
- There are variations of Binary Heap like Fibonacci Heap that can support insert and decrease-key in $\Theta(1)$ time

Is Binary Heap always better?
Although Binary Heap is for Priority Queue, BSTs have their own advantages and the list of advantages is in-fact bigger compared to binary heap.

- Searching an element in self-balancing BST is O(Logn) which is O(n) in Binary Heap.
- We can print all elements of BST in sorted order in O(n) time, but Binary Heap requires O(nLogn) time.
- Floor and ceil can be found in O(Logn) time.
- K'th largest/smallest elementbe found in O(Logn) time by augmenting tree with an additional field.

This article is contributed by **Vivek Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/why-is-binary-heap-preferred-over-bst-for-priority-queue/

Chapter 2

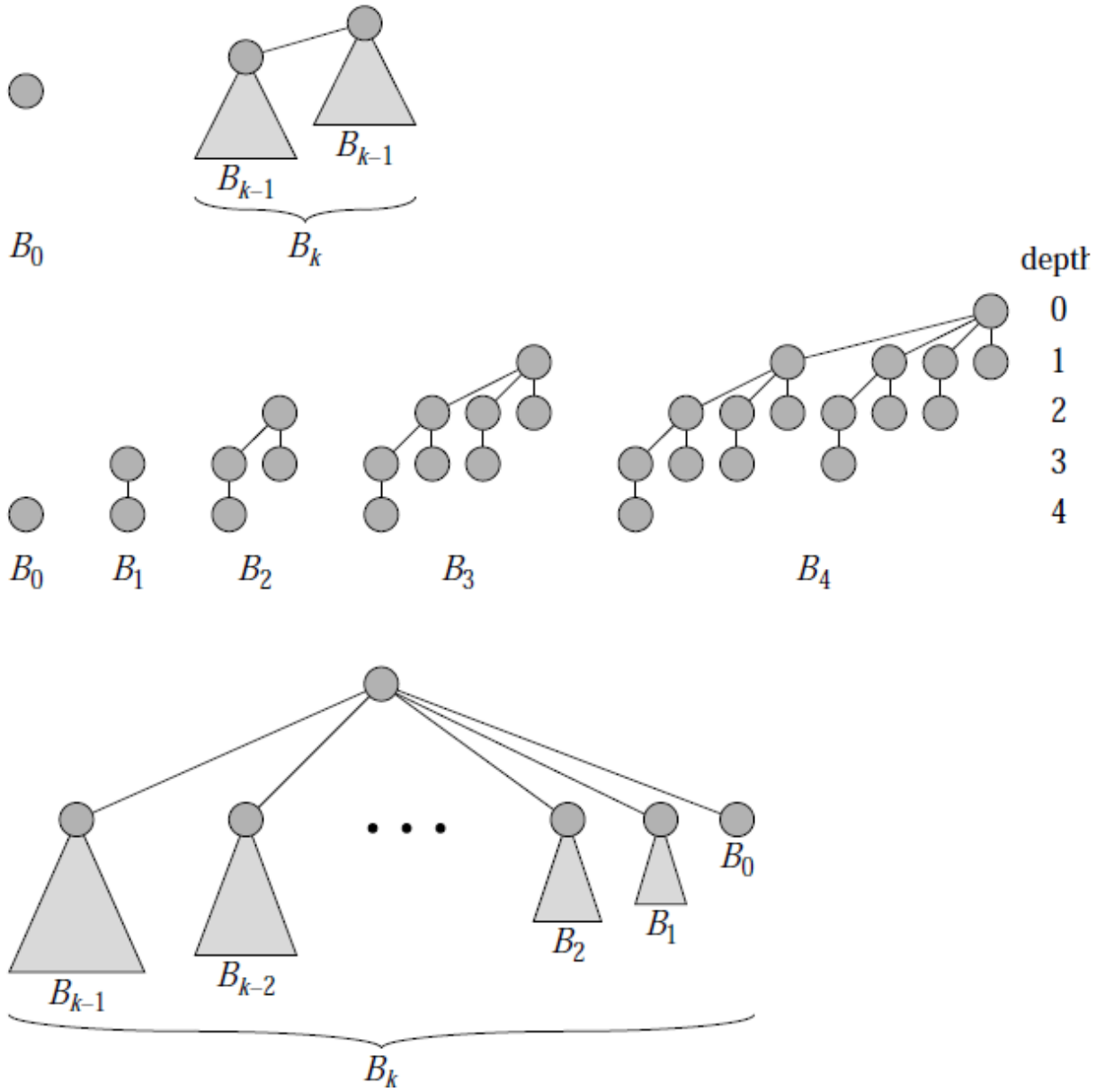
Binomial Heap

The main application of Binary Heapis as implement priority queue. Binomial Heap is an extension of Binary Heapthat provides faster union or merge operation together with other operations provided by Binary Heap.

A Binomial Heap is a collection of Binomial Trees

What is a Binomial Tree?
A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1, and making one as leftmost child of other.
A Binomial Tree of order k has following properties.
a) It has exactly 2^k nodes.
b) It has depth as k.
c) There are exactly kCi nodes at depth i for i = 0, 1, . . . , k.
d) The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

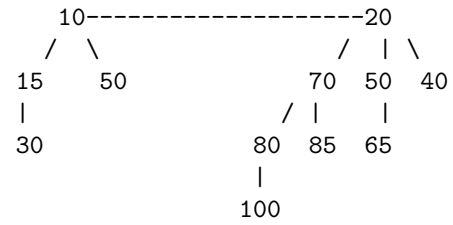
The following diagram is taken from 2nd Edition of CLRS book.



Binomial Heap:
A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows Min Heap property. And there can be at-most one Binomial Tree of any degree.

Examples Binomial Heap:

```
 12------------10--------------------20
             /  \                  /  | \
           15    50              70   50  40
           |                   / |    |
           30                80  85   65
                             |
                            100
A Binomial Heap with 13 nodes. It is a collection of 3
Binomial Trees of orders 0, 2 and 3 from left to right.
```



A Binomial Heap with 12 nodes. It is a collection of 2
Binomial Trees of orders 2 and 3 from left to right.

Binary Representation of a number and Binomial Heaps
A Binomial Heap with n nodes has number of Binomial Trees equal to the number of set bits in Binary representation of n. For example let n be 13, there 3 set bits in binary representation of n (00001101), hence 3 Binomial Trees. We can also relate degree of these Binomial Trees with positions of set bits. With this relation we can conclude that there are O(Logn) Binomial Trees in a Binomial Heap with 'n' nodes.

Operations of Binomial Heap:
The main operation in Binomial Heap is union(), all other operations mainly use this operation. The union() operation is to combine two Binomial Heaps into one. Let us first discuss other operations, we will discuss union later.

1) insert(H, k): Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.

2) getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires O(Logn) time. It can be optimized to O(1) by maintaining a pointer to minimum key root.

3) extractMin(H): This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap. This operation requires O(Logn) time.

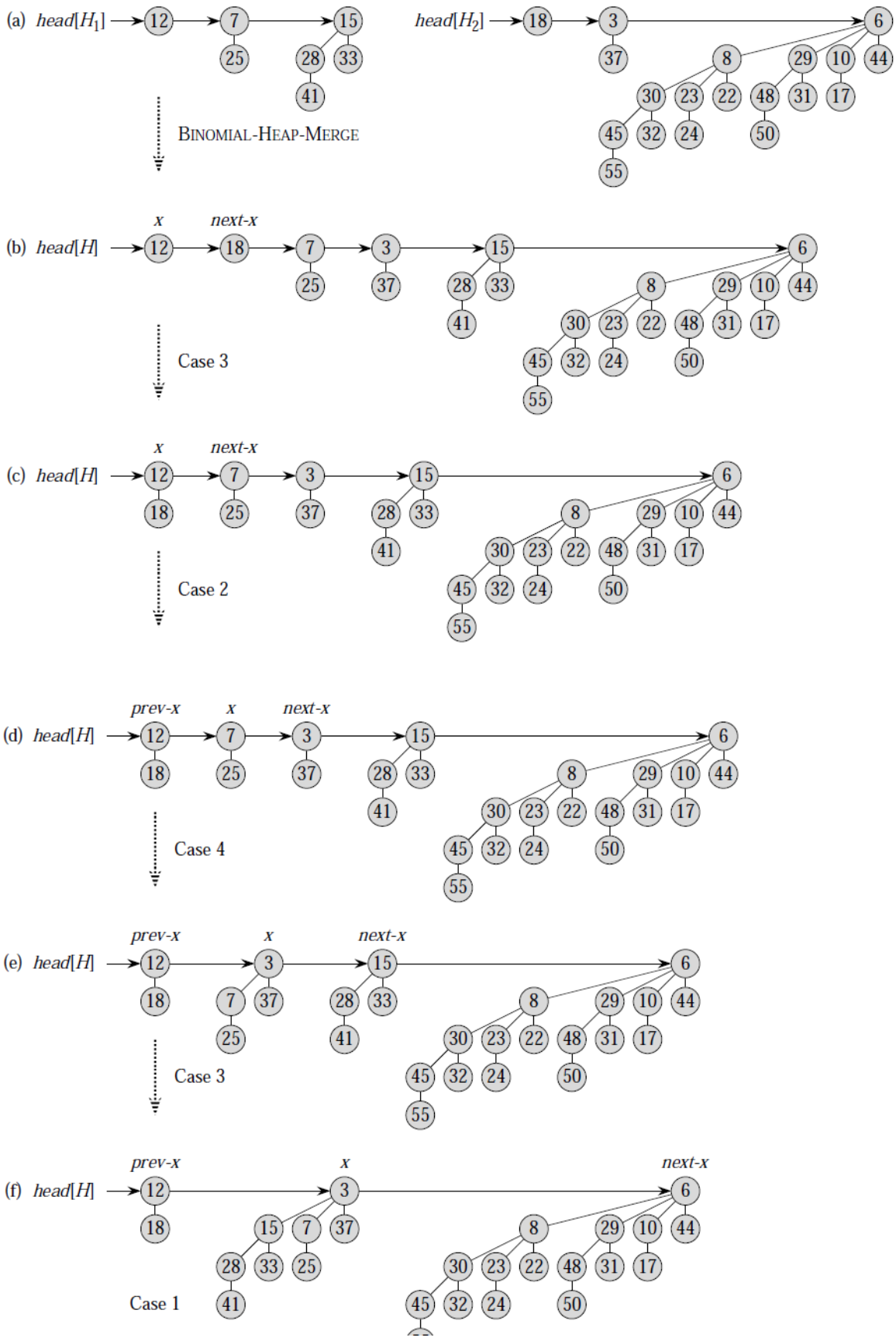
4) delete(H): Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extract-Min().

5) decreaseKey(H): decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node. Time complexity of decreaseKey() is O(Logn).

Union operation in Binomial Heap:
Given two Binomial Heaps H1 and H2, union(H1, H2) creates a single Binomial Heap.
1) The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.

2) After the simple merge, we need to make sure that there is at-most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of same order. We traverse the list of merged roots, we keep track of three pointers, prev, x and next-x. There can be following 4 cases when we traverse the list of roots.
——Case 1: Orders of x and next-x are not same, we simply move ahead.
In following 3 cases orders of x and next-x are same.
——Case 2: If order of next-next-x is also same, move ahead.
——Case 3: If key of x is smaller than or equal to key of next-x, then make next-x as a child of x by linking it with x.
——Case 4: If key of x is greater, then make x as child of next.

The following diagram is taken from 2nd Edition of CLRS book.


(a) head[H1]
head[H2]
BINOMIAL-HEAP-MERGE
(b) head[H]
x
next-x
Case 3
(c) head[H]
x
next-x
Case 2
(d) head[H]
prev-x
x
next-x
Case 4
(e) head[H]
prev-x
x
next-x
Case 3
(f) head[H]
prev-x
x
next-x
Case 1

How to represent Binomial Heap?
A Binomial Heap is a set of Binomial Trees. A Binomial Tree must be represented in a way that allows sequential access to all siblings, starting from the leftmost sibling (We need this in and extractMin() and delete()). The idea is to represent Binomial Trees as leftmost child and right-sibling representation, i.e., every node stores two pointers, one to the leftmost child and other to the right sibling.

We will soon be discussing implementation of Binomial Heap.

Sources:

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/binomial-heap-2/

Chapter 3

Fibonacci Heap | Set 1 (Introduction)

Heaps are mainly used for implementing priority queue. We have discussed below heaps in previous posts.

Binary Heap
Binomial Heap

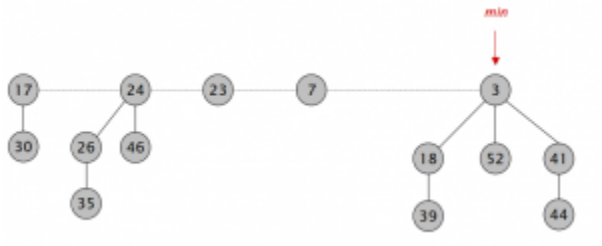
In terms of Time Complexity, Fibonacci Heap beats both Binary and Binomial Heaps.

Below are amortized time complexities of **Fibonacci Heap**.

```
 1) Find Min:      Θ(1)     [Same as both Binary and Binomial]
2) Delete Min:    O(Log n) [Θ(Log n) in both Binary and Binomial]
3) Insert:        Θ(1)     [Θ(Log n) in Binary and Θ(1) in Binomial]
4) Decrease-Key:  Θ(1)     [Θ(Log n) in both Binary and Binomial]
5) Merge:         Θ(1)     [Θ(m Log n) or Θ(m+n) in Binary and
                            Θ(Log n) in Binomial]
```

Like Binomial Heap, Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be Binomial Tree).

Below is an example Fibonacci Heap taken from here.



Fibonacci Heap maintains a pointer to minimum value (which is root of a tree). All tree roots are connected using circular doubly linked list, so all of them can be accessed using single 'min' pointer.

The main idea is to execute operations in "lazy" way. For example merge operation simply links two heaps, insert operation simply adds a new tree with single node. The operation extract minimum is the most complicated operation. It does delayed work of consolidating trees. This makes delete also complicated as delete first decreases key to minus infinite, then calls extract minimum.

Below are some interesting facts about Fibonacci Heap

1. The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With Binary Heap, time complexity of these algorithms is O(VLogV + ELogV). If Fibonacci Heap is used, then time complexity is improved to O(VLogV + E)
2. Although Fibonacci Heap looks promising time complexity wise, it has been found slow in practice as hidden constants are high (Source Wiki).
3. Fibonacci heap are mainly called so because Fibonacci numbers are used in the running time analysis. Also, every node in Fibonacci Heap has degree at most O(log n) and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the kth Fibonacci number.

We will soon be discussing Fibonacci Heap operations in detail.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/

Chapter 4

k largest(or smallest) elements in an array | added Min Heap method

Question: Write an efficient program for printing k largest elements in an array. Elements in array can be in any order.

For example, if given array is [1, 23, 12, 9, 30, 2, 50] and you are asked for the largest 3 elements i.e., k = 3 then your program should print 50, 30 and 23.

Method 1 (Use Bubble k times)
Thanks to Shailendra for suggesting this approach.
1) Modify Bubble Sort to run the outer loop at most k times.
2) Print the last k elements of the array obtained in step 1.

Time Complexity: O(nk)

Like Bubble sort, other sorting algorithms like Selection Sort can also be modified to get the k largest elements.

Method 2 (Use temporary array)
K largest elements from arr[0..n-1]

1) Store the first k elements in a temporary array temp[0..k-1].
2) Find the smallest element in temp[], let the smallest element be *min*.
3) For each element *x* in arr[k] to arr[n-1]
If *x* is greater than the min then remove *min* from temp[] and insert *x*.
4) Print final k elements of *temp[]*

Time Complexity: O((n-k)*k). If we want the output sorted then O((n-k)*k + klogk)

Thanks to nesamani1822 for suggesting this method.

Method 3(Use Sorting)
1) Sort the elements in descending order in O(nLogn)
2) Print the first k numbers of the sorted array O(k).

Time complexity: O(nlogn)

Method 4 (Use Max Heap)
1) Build a Max Heap tree in O(n)
2) Use Extract Max k times to get k maximum elements from the Max Heap O(klogn)

Time complexity: O(n + klogn)

Method 5(Use Oder Statistics)
1) Use order statistic algorithm to find the kth largest element. Please see the topic selection in worst-case linear time O(n)
2) Use QuickSort Partition algorithm to partition around the kth largest number O(n).
3) Sort the k-1 elements (elements greater than the kth largest element) O(kLogk). This step is needed only if sorted output is required.

Time complexity: O(n) if we don't need the sorted output, otherwise O(n+kLogk)

Thanks to Shilpifor suggesting the first two approaches.

Method 6 (Use Min Heap)
This method is mainly an optimization of method 1. Instead of using temp[] array, use Min Heap.

Thanks to geek4ufor suggesting this method.

1) Build a Min Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. O(k)

2) For each element, after the kth element (arr[k] to arr[n-1]), compare it with root of MH.
.....a) If the element is greater than the root then make it root and call heapifyfor MH
.....b) Else ignore it.
// The step 2 is O((n-k)*logk)

3) Finally, MH has k largest elements and root of the MH is the kth largest element.

Time Complexity: O(k + (n-k)Logk) without sorted output. If sorted output is needed then O(k + (n-k)Logk + kLogk)

All of the above methods can also be used to find the kth largest (or smallest) element.

Please write comments if you find any of the above explanations/algorithms incorrect, or find better ways to solve the same problem.

References:
http://en.wikipedia.org/wiki/Selection_algorithm

Asked by geek4u

Source

http://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/

Chapter 5

Sort a nearly sorted (or K sorted) array

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in O(n log k) time.
For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Source: Nearly sorted algorithm

We can **use Insertion Sort** to sort the elements efficiently. Following is the C code for standard Insertion Sort.

```
/* Function to sort an array using insertion sort*/
void insertionSort(int A[], int size)
{
   int i, key, j;
   for (i = 1; i < size; i++)
   {
       key = A[i];
       j = i-1;

       /* Move elements of A[0..i-1], that are greater than key, to one
          position ahead of their current position.
          This loop will run at most k times */
       while (j >= 0 && A[j] > key)
       {
           A[j+1] = A[j];
           j = j-1;
       }
       A[j+1] = key;
   }
}
```

The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall *complexity will be O(nk)*

We can sort such arrays **more efficiently with the help of Heap data structure**. Following is the detailed process that uses Heap.
1) Create a Min Heap of size k+1 with first k+1 elements. This will take O(k) time (See this GFact)
2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take Logk time. So overall complexity will be O(k) + O((n-k)*logK)

```
#include<iostream>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor
    MinHeap(int a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int );

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to remove min (or root), add a new value x, and return old root
    int replaceMin(int x);

    // to extract the root which is the minimum element
    int extractMin();
};

// Given an array of size n, where every element is k away from its target
// position, sorts the array in O(nLogk) time.
int sortK(int arr[], int n, int k)
{
    // Create a Min Heap of first (k+1) elements from
    // input array
    int *harr = new int[k+1];
    for (int i = 0; i<=k && i<n; i++) // i < n condition is needed when k > n
        harr[i] = arr[i];
    MinHeap hp(harr, k+1);

    // i is index for remaining elements in arr[] and ti
    // is target index of for cuurent minimum element in
    // Min Heapm 'hp'.
```

```
    for(int i = k+1, ti = 0; ti < n; i++, ti++)
    {
        // If there are remaining elements, then place
        // root of heap at target index and add arr[i]
        // to Min Heap
        if (i < n)
            arr[ti] = hp.replaceMin(arr[i]);

        // Otherwise place root at its target index and
        // reduce heap size
        else
            arr[ti] = hp.extractMin();
    }
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int a[], int size)
{
    heap_size = size;
    harr = a;  // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    int root = harr[0];
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        heap_size--;
        MinHeapify(0);
    }
    return root;
}

// Method to change root with given value x, and return the old root
int MinHeap::replaceMin(int x)
{
    int root = harr[0];
    harr[0] = x;
    if (root < x)
        MinHeapify(0);
    return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
```

```
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
   for (int i=0; i < size; i++)
       cout << arr[i] << " ";
   cout << endl;
}

// Driver program to test above functions
int main()
{
    int k = 3;
    int arr[] = {2, 6, 3, 12, 56, 8};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortK(arr, n, k);

    cout << "Following is sorted array\n";
    printArray (arr, n);

    return 0;
}
```

Output:

```
 Following is sorted array
2 3 6 8 12 56
```

The Min Heap based method takes O(nLogk) time and uses O(k) auxiliary space.

We can also **use a Balanced Binary Search Tree** instead of Heap to store K+1 elements. The insertand deleteoperations on Balanced BST also take O(Logk) time. So Balanced BST based method will also take O(nLogk) time, but the Heap bassed method seems to be more efficient as the minimum element will always be at root. Also, Heap doesn't need extra space for left and right pointers.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/nearly-sorted-algorithm/

Category: Arrays

Post navigation

← Adobe Interview | Set 1 Microsoft Interview | Set 4 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 6

Tournament Tree (Winner Tree) and Binary Heap

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

Tournament tree is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

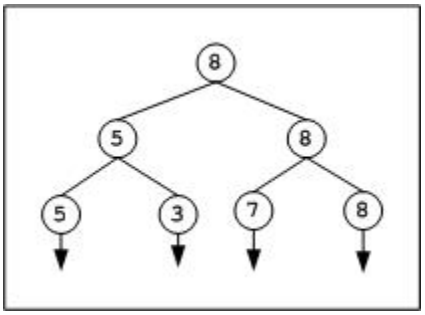
There will be N – 1 internal nodes in a binary tree with N leaf (external) nodes. For details see this post (put n = 2 in equation given in the post).

It is obvious that to select the best player among N players, (N – 1) players to be eliminated, i.e. we need minimum of (N – 1) games (comparisons). Mathematically we can prove it. In a binary tree I = E – 1, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need N – 1 (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in $\mathbf{(N + \log_2 N - 2)}$ comparisons. For details read this comment.

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

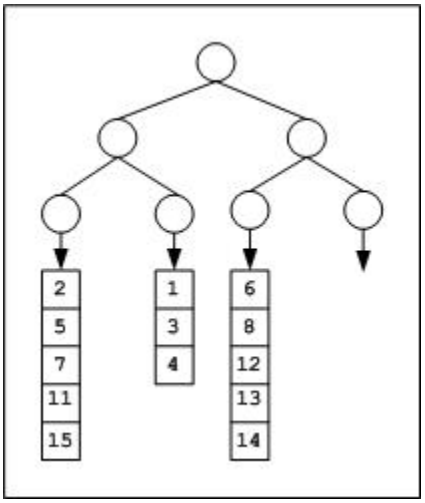
Median of Sorted Arrays

Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL ($\log_2$M)** to have atleast M external nodes.

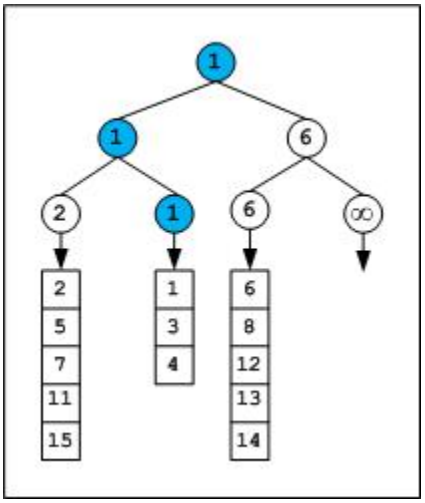
Consider an example. Given 3 (M = 3) sorted integer arrays of maximum size 5 elements.

```
 { 2, 5, 7, 11, 15 } ---- Array1
{1, 3, 4} ---- Array2
{6, 8, 12, 13, 14} ---- Array3
```

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3 .= 1.585 = 2$ rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



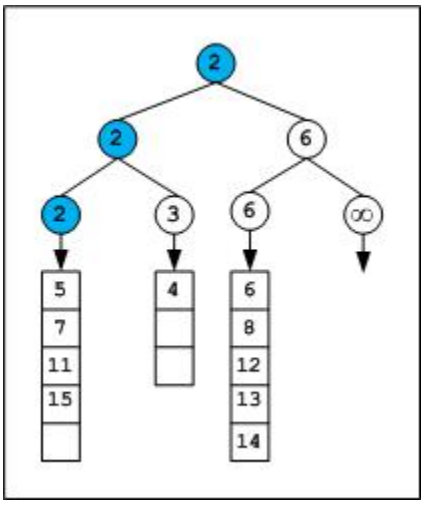
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is (5+3+5)/2 = 7th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size L_1, L_2 … L_m requires time complexity of $O((L_1 + L_2 + \ldots + L_m) * logM)$ to merge all the arrays, and $O(m*logM)$ time to find median, where m is median position.

Select smallest one million elements from one billion unsorted elements:

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree (heap) in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to $2^k - 1$ where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. We will have code in an upcoming article.

Related Posts

Link 1, Link 2, Link 3, Link 4, Link 5, Link 6, Link 7.

— by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/tournament-tree-and-binary-heap/