Course Name: Data Structures and Applications

Course Code: BCS304

Module 5
Hashing, Priority Queues, BST

Hashing

- The time required to search for an key in other searching techniques depends on the number of elements in the collection
- Hashing or hash addressing is independent of the number n

Terminology

- Hashing will be oriented towards file management
- We assume that there is a file F on n records with s set K of keys which uniquely determine the records in F
- F is maintained in memory by a table T of m memory locations and L is the set of memory addresses of the locations in T

- The general idea is to use the key to determine the address of a record, but care must be taken so that a great deal of space is not wasted
- This takes a form of a function H from the set K of keys into the set L of memory addresses.
- Such a function H: K→L is called a hash function or hashing function

- Hash function may not yield distinct values : it is possible that two different keys k1 and k2 will yield the same hash address. This situation is called collision and some method must be used to resolve it
- Hash function
- Collision resolutions

Hash functions

- Two principle criteria used in selecting a hash function H: K→L are:
- The function H should be very easy and quick to compute
- The function H should as far as possible, uniformly distribute the hash addresses throughout the set L so that there are minimum number of collisions

Static hashing

- Division
- Mid-Square
- Folding
- Digit Analysis

Converting keys into integers

```
Unsigned int stringToInt (char *key)
{
Int number=0;
While(*key)
Number += *key++;
Return number;
}
```

- Simple additive approach

```
Unsigned int StingToInt ( char *key)
{
Int number = 0;
While ( *key)
{
Number += *key++;
If(*key) number +=((int) *key++) <<8;
}
Retunr number;
}
```

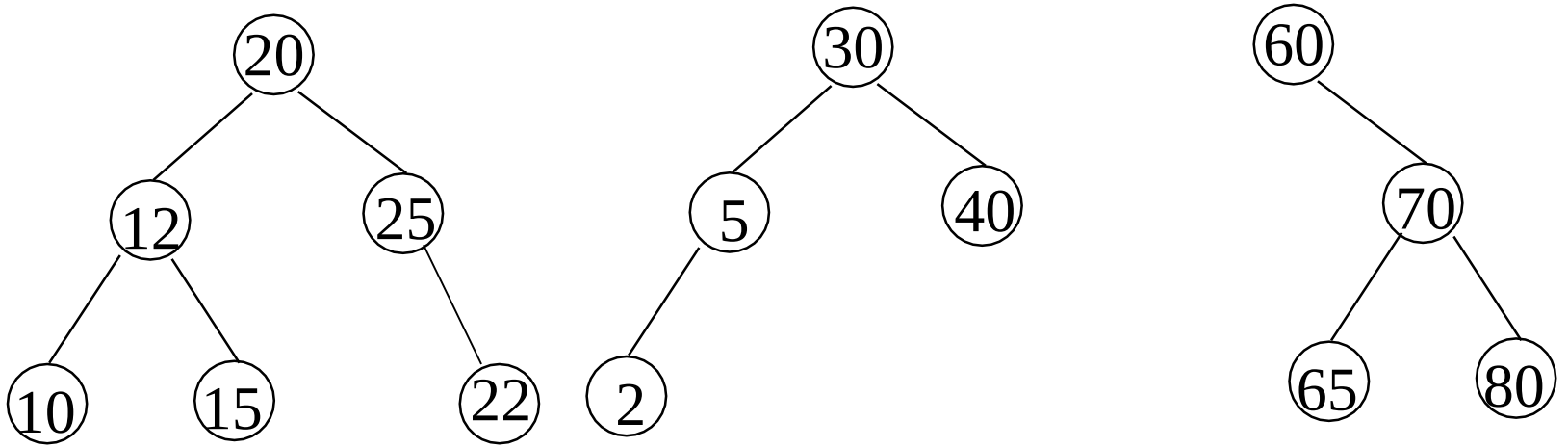
Overflow handling

- Open addressing
- Chaining
- Four open addressing methods
- Linear probing or linear open addressing
- Quadratic probing
- Rehashing
- Random probing

Binary Search Tree

- Heap
  - a min (max) element is deleted. $O(\log_2 n)$
  - deletion of an arbitrary element $O(n)$
  - search for an arbitrary element $O(n)$
- Binary search tree
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.

Examples of Binary Search Trees



Searching a Binary Search Tree

```
tree_pointer search(tree_pointer root,
                                int key)
{
/* return a pointer to the node that
  contains key. If there is no such  node,
  return NULL */

   if (!root) return NULL;
   if (key == root->data) return root; if
   (key < root->data)
        return search(root->left_child,
                                key);
   return search(root->right_child,key);
}
```

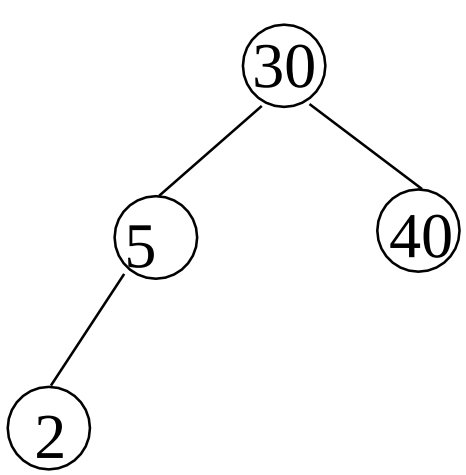
Another Searching Algorithm

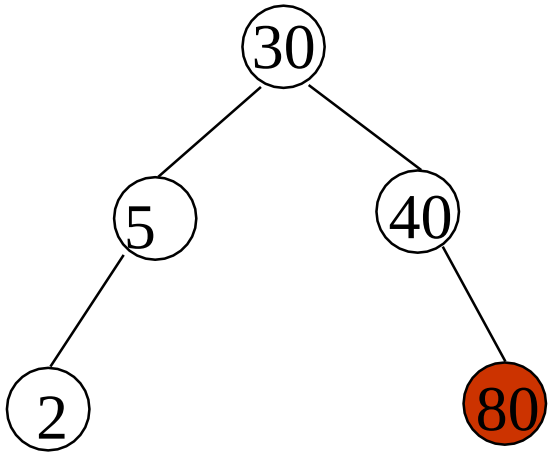
```
tree_pointer search2(tree_pointer tree, int
  key)
{
   while (tree) {
      if (key == tree->data) return tree; if
      (key < tree->data)
           tree = tree->left_child;  else
      tree = tree->right_child;


   }
   return NULL;
}
```

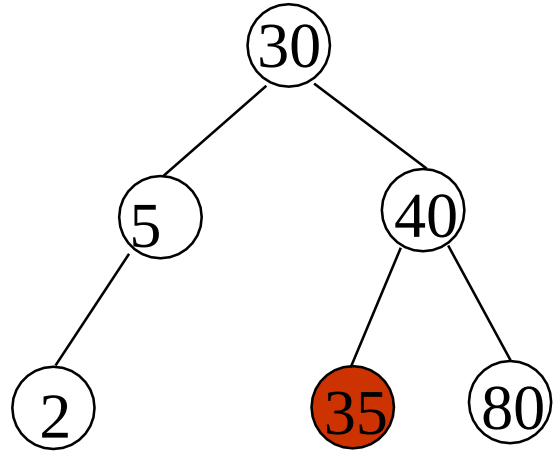
O(h)

Insert Node in Binary Search Tree





Insert 80

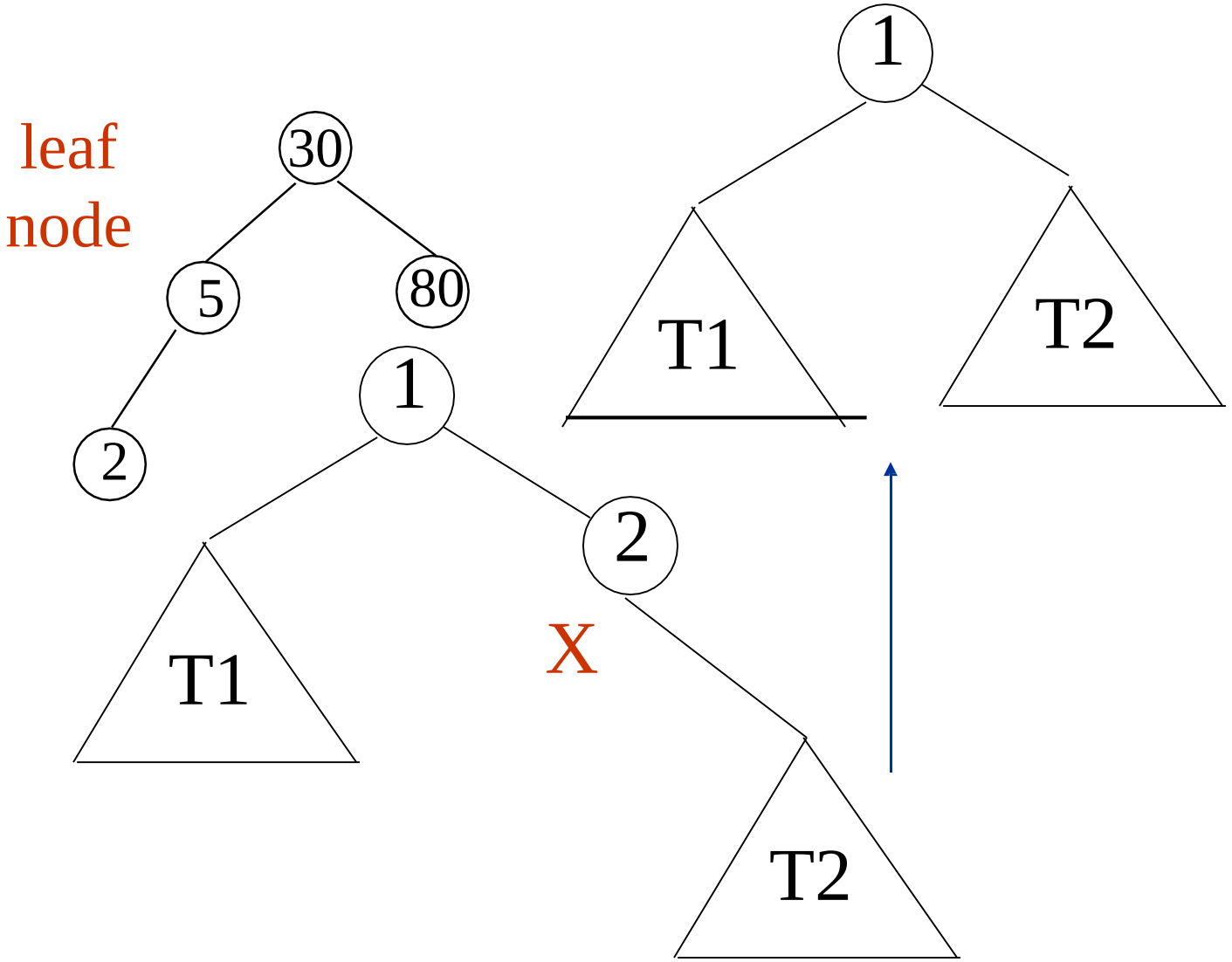


Insert 35

Insertion into A Binary Search Tree

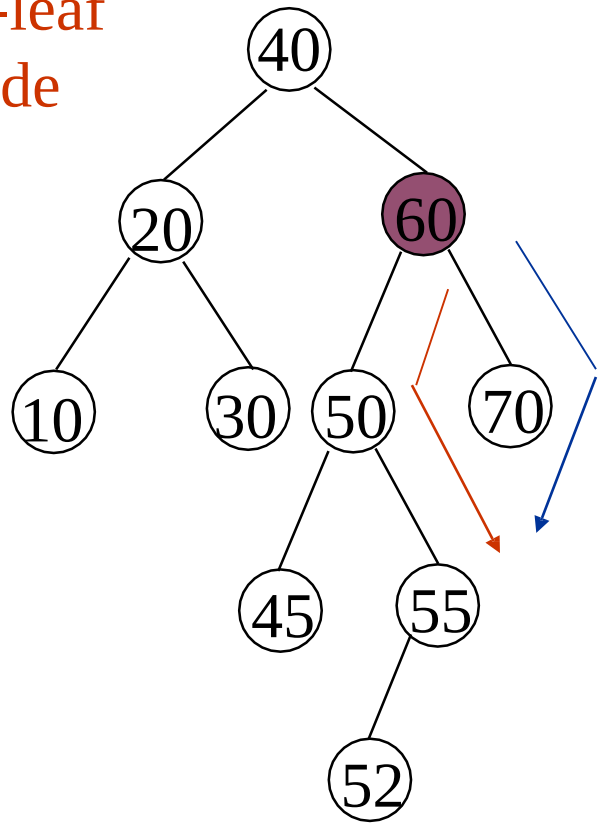
```
void insert_node(tree_pointer *node, int num)
{tree_pointer ptr,
        temp = modified_search(*node, num); if (temp ||
   !(*node)) {
    ptr = (tree_pointer) malloc(sizeof(node)); if (IS_FULL(ptr)) {
       fprintf(stderr, "The memory is full\n");
       exit(1);
    }
    ptr->data = num;
    ptr->left_child = ptr->right_child = NULL;
    if (*node)
       if (num<temp->data) temp->left_child=ptr; else temp-
           >right_child = ptr;
    else *node = ptr;
   }
}
```

Deletion for A Binary Search Tree

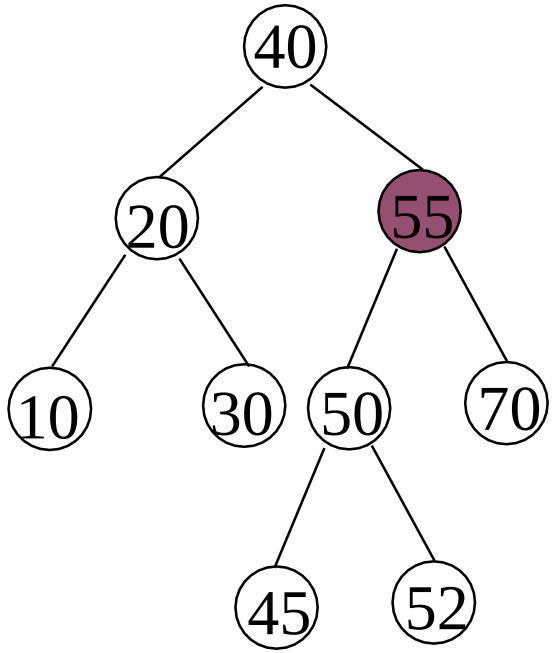


Deletion for A Binary Search Tree

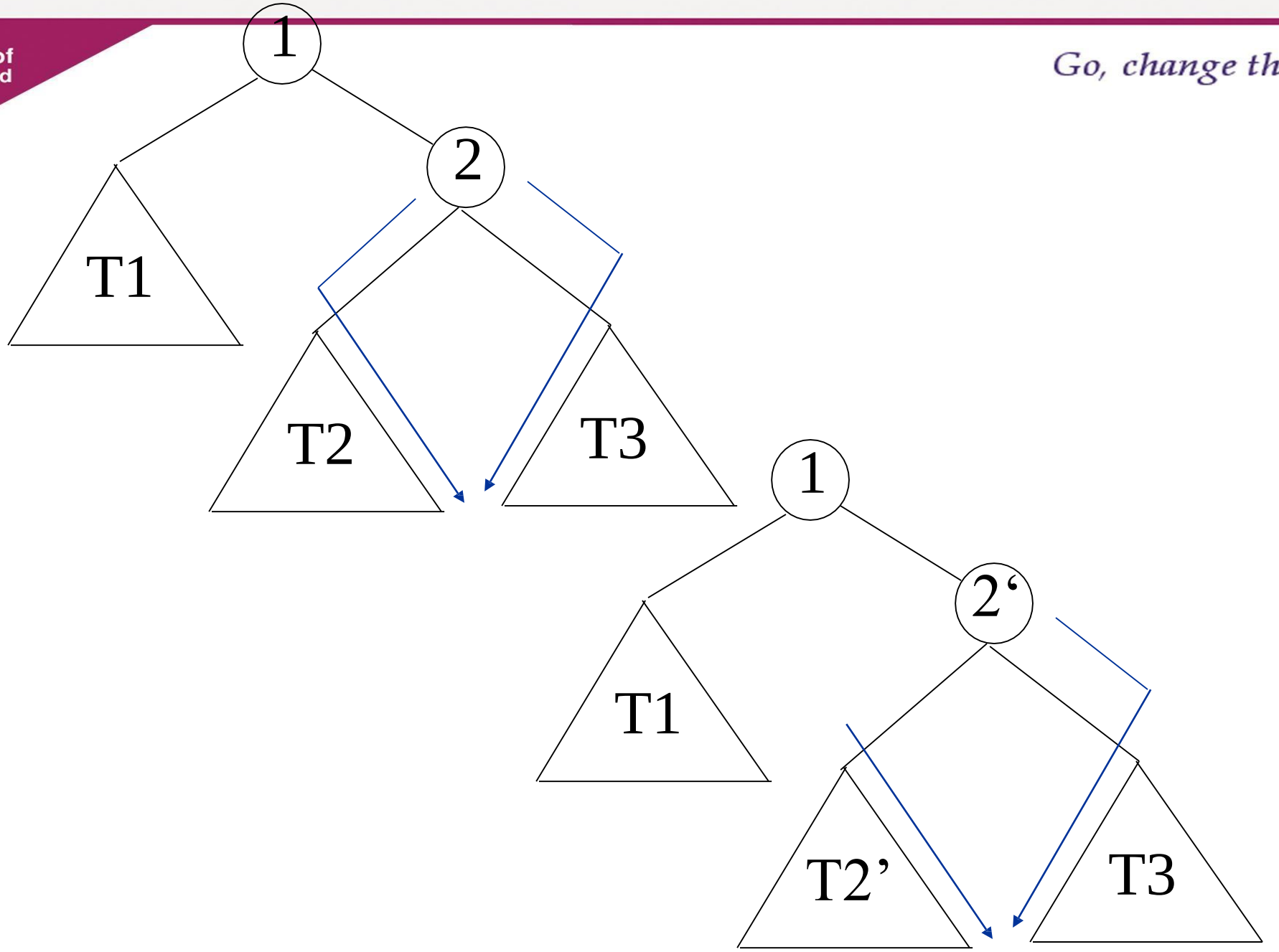
non-leaf node



Before deleting 60



After deleting 60

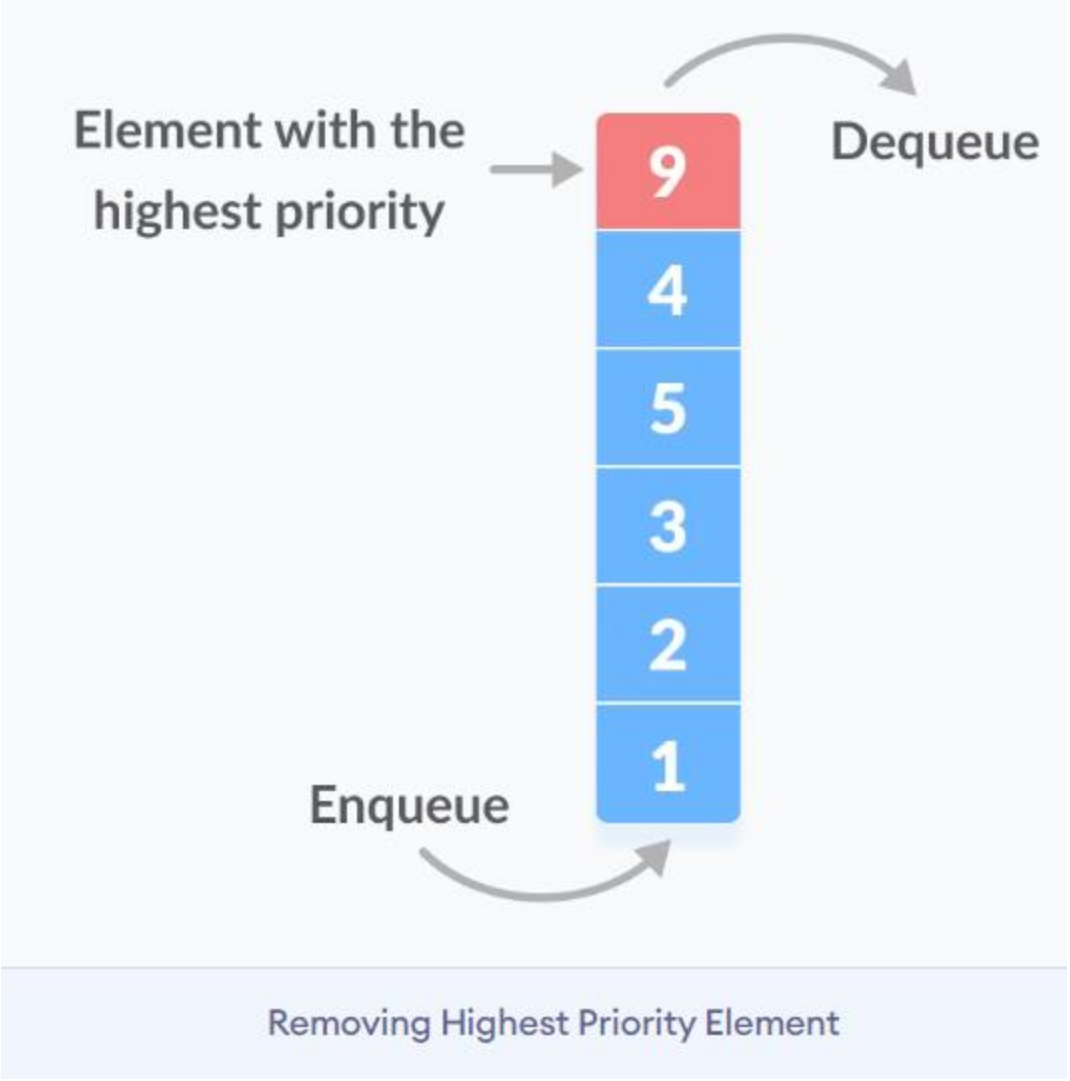

1
2
T1
T2
T3
1
2‘
T1
T2’
T3

Priority Queue

- A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.
- However, if elements with the same priority occur, they are served according to their order in the queue.

Assigning Priority Value

- Generally, the value of the element itself is considered for assigning the priority. For example,
- The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.
- We can also set priorities according to our needs.



Difference between Priority Queue and Normal Queue

In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.

Implementation of Priority Queue

- Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.
- Hence, we will be using the heap data structure to implement the priority queue in this tutorial. A max-heap is implemented in the following operations. If you want to learn more about it, please visit max-heap and min-heap.

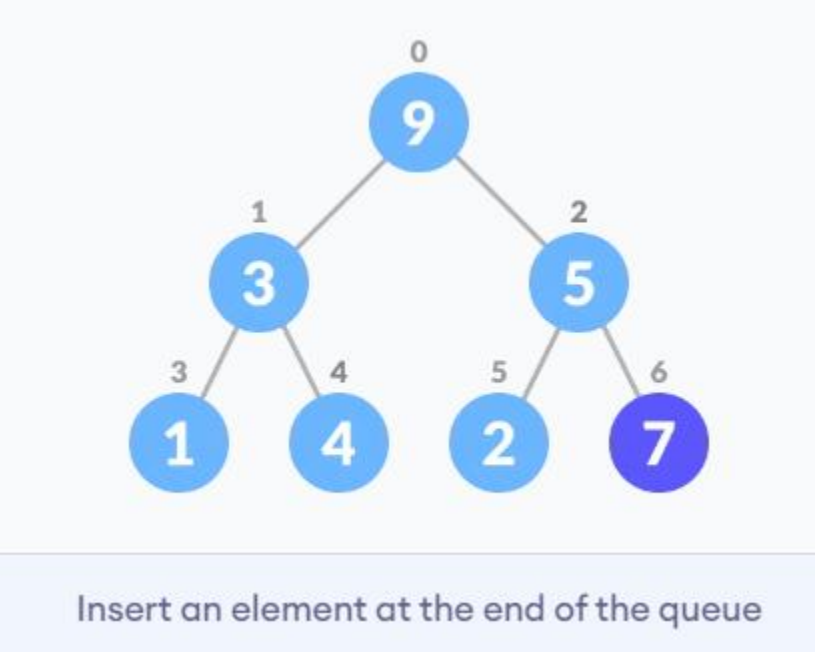
Operations	peek	insert	delete
Linked List	O(1)	O(n)	O(1)
Binary Heap	O(1)	O(log n)	O(log n)
Binary Search Tree	O(1)	O(log n)	O(log n)

Priority Queue Operations

Inserting an Element into the Priority Queue

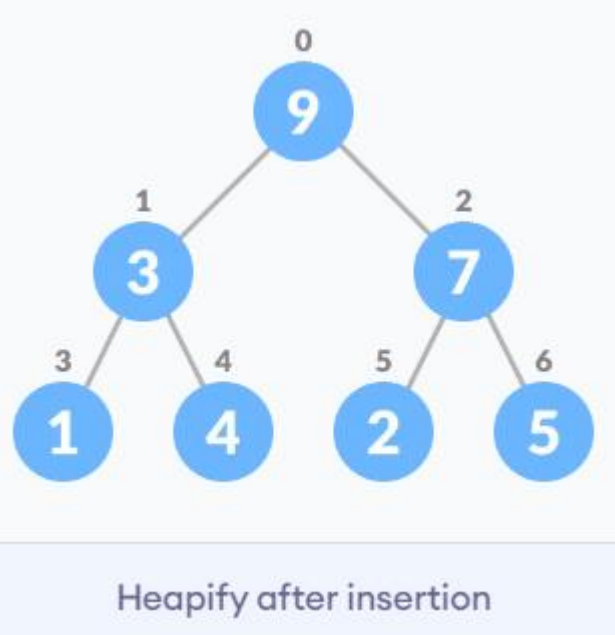
Inserting an element into a priority queue (max-heap) is done by the following steps.

Insert the new element at the end of the tree.



Insert an element at the end of the queue

Heapify the tree.

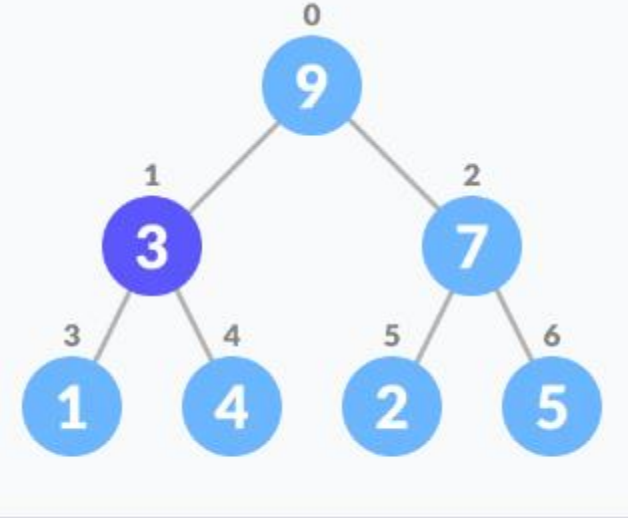


Heapify after insertion

Deleting an Element from the Priority Queue

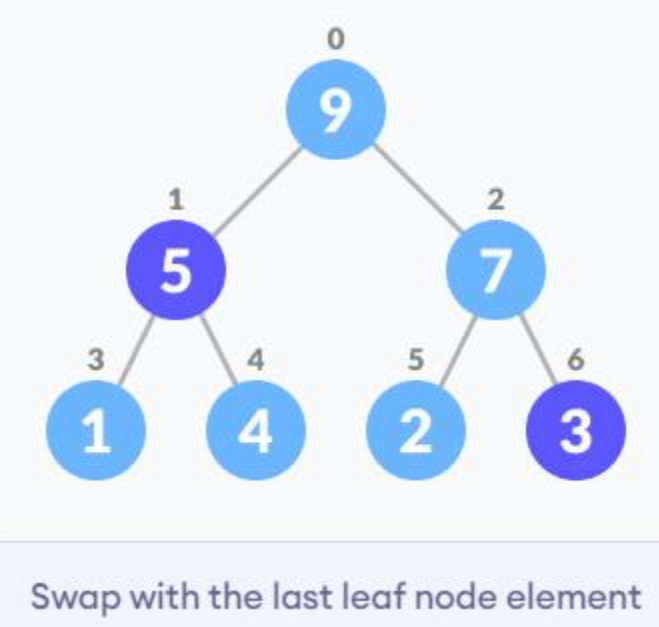
Deleting an element from a priority queue (max-heap) is done as follows:

•Select the element to be deleted.



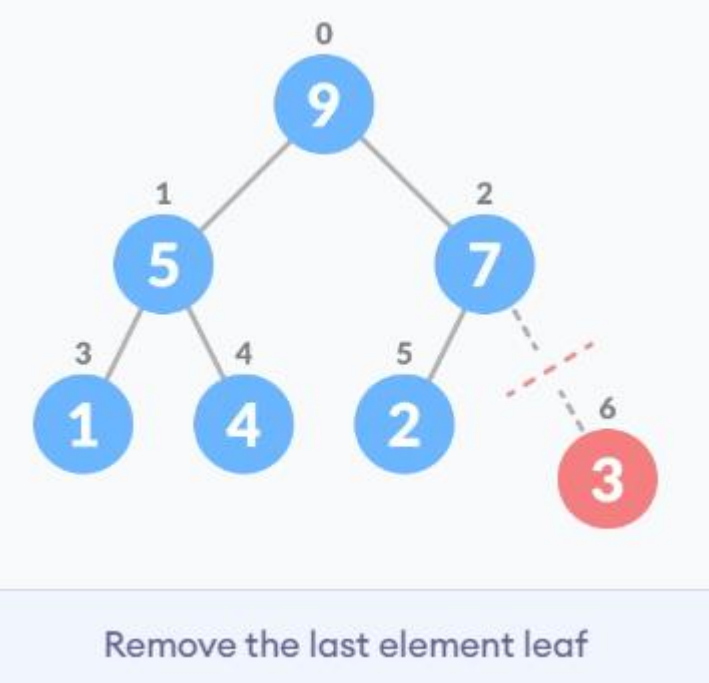
Select the element to be deleted

Swap it with the last element.



Swap with the last leaf node element

Remove the last element.



Remove the last element leaf

```
// Priority Queue implementation in C

#include <stdio.h>
int size = 0;
void swap(int *a, int *b) {
  int temp = *b;
  *b = *a;
  *a = temp;
}
```

```
// Function to heapify the tree
void heapify(int array[], int size, int i) {
  if (size == 1) {
    printf("Single element in the heap");
  } else {
    // Find the largest among root, left child and right child
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < size && array[l] > array[largest])
      largest = l;
    if (r < size && array[r] > array[largest])
      largest = r;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
      swap(&array[i], &array[largest]);
      heapify(array, size, largest);
    }
  }
}
```

```
// Function to insert an element into the tree
void insert(int array[], int newNum) {
  if (size == 0) {
    array[0] = newNum;
    size += 1;
  } else {
    array[size] = newNum;
    size += 1;
    for (int i = size / 2 - 1; i >= 0; i--) {
      heapify(array, size, i);
    }
  }
}
```

```
// Function to delete an element from the tree
void deleteRoot(int array[], int num) {
  int i;
  for (i = 0; i < size; i++) {
    if (num == array[i])
      break;
  }

  swap(&array[i], &array[size - 1]);
  size -= 1;
  for (int i = size / 2 - 1; i >= 0; i--) {
    heapify(array, size, i);
  }
}

// Print the array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i)
    printf("%d ", array[i]);
  printf("\n");
}
```

```
int main() {
  int array[10];

  insert(array, 3);
  insert(array, 4);
  insert(array, 9);
  insert(array, 5);
  insert(array, 2);

  printf("Max-Heap array: ");
  printArray(array, size);

  deleteRoot(array, 4);

  printf("After deleting an element: ");

  printArray(array, size);
}
```

Double Ended Priority Queue

Introduction to the double-ended priority queue

A double-ended priority queue (DEPQ) is a data structure that stores a collection of elements, where each element is associated with a priority or value. Elements can be inserted and removed from both ends of the queue based on their priority.

The highest priority element can be accessed from either end of the queue without removing it. In a DEPQ, the elements are stored in a way that maintains their priority order, such that the highest priority element is always at the front or back of the queue, depending on the implementation.

*Priority Queue - A **priority queue** is a collection of items where each item has a priority assigned to it, and the items are handled in priority order.*

*Double-Ended Queue - A **double-ende queue** is a collection of items that permits insertion and deletion from both the front and the back of the queue*

Properties of DEPQ

- A DEPQ permits actions like inserting and deleting items with priorities at both ends of the queue.
- In spite of where they are in the queue, the elements with higher priorities are always processed first.
- The DEPQ is a valuable data structure for scheduling, task management, and resource allocation applications because of this.
- Several data structures, including binary heaps, Fibonacci heaps, and balanced binary search trees, can be used to build DEPQs.
- The needs of the particular application and the trade-offs between time and space complexity determine the best data structure to use.

Operations are performed on a double-ended priority queue (DEPQ):

- **getMin():** This function returns the minimum element.
- **getMax():** This function returns the maximum element
- **put(x):** This function inserts the element '*x*' into the DEPQ.
- **removeMin():** As the name indicates, it removes the minimum element from DEPQ
- **removeMax():** It removes the maximum element from the DEPQ.

Self-balancing BST implementation of a DEPQ

- A self-balancing BST implementation of a DEPQ provides effective time complexity for all of its operations, and its space complexity is comparable to that of other implementations. Self-balancing trees automatically retain their balanced structure, ensuring quick and reliable performance while working with big data sets.
- *In C++, a self-balancing binary search tree, such as red-black or AVL trees, is commonly used to implement the set container. These data structures offer set operations with effective worst-case and average-case time complexity.*

```
// C++ program to implement double-ended
// priority queue using self-balancing BST
#include <iostream>
#include <set>

using namespace std;

// Defining DEPQ class
class DEPQ
{
private:
   set<int> s;

public:
   // Insert a value into the DEPQ
   // Time Complexity = O(Log n)
   void put(int val)
   {
      s.insert(val);
   }

   // Get the minimum value in the DEPQ
   // Time Complexity = O(1)
   int getMin()
   {
      return *s.begin();
   }

   // Get the maximum value in the DEPQ
   // Time Complexity = O(1)
   int getMax()
   {
      return *prev(s.end());
   }

   // Delete the minimum value from the DEPQ
   // Time Complexity = O(Log n)
   void removeMin()
   {
      if (!s.empty())
      {
         s.erase(s.begin());
      }
   }
```

```
// Delete the maximum value from the DEPQ
  // Time Complexity = O(Log n)
  void removeMax()
  {
     if (!s.empty())
     {
        s.erase(prev(s.end()));
     }
  }

  // Get the size of the DEPQ
  // Time Complexity = O(1)
  int size()
  {
     return s.size();
  }

// Check if the DEPQ is empty
   // Time Complexity = O(1)
   bool isEmpty()
   {
      return s.empty();
   }
};
```

```
int main()
{
    DEPQ d;

    int choice, val;

    // Printing Options
    cout << "\nDEPQ Operations:" << endl;
    cout << "1. Insert an Element" << endl;
    cout << "2. Get Minimum Element" << endl;
    cout << "3. Get Maximum Element" << endl;
    cout << "4. Delete Minimum Element" << endl;
    cout << "5. Delete Maximum Element" << endl;
    cout << "6. Size" << endl;
    cout << "7. Is Empty?" << endl;
    cout << "0. Exit" << endl;
}
```