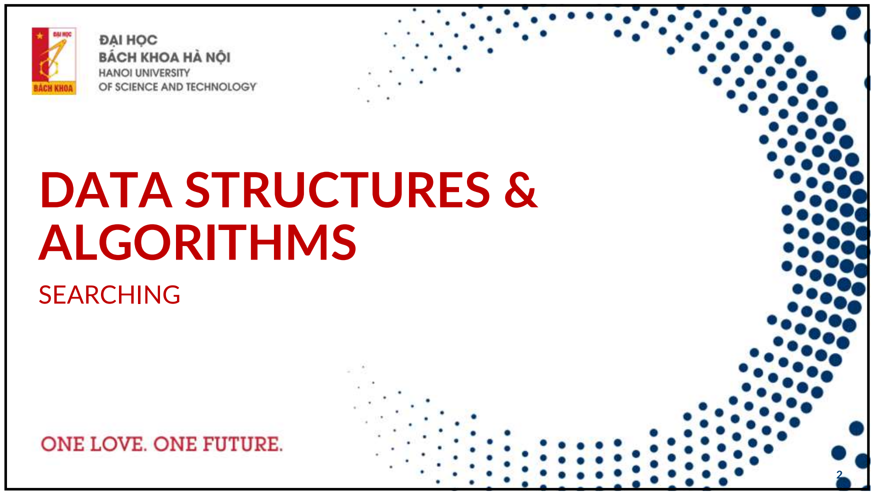
HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

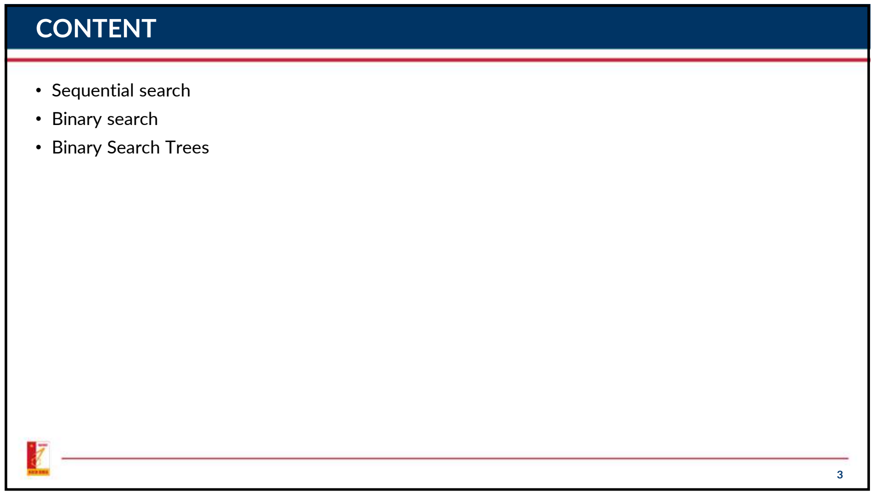

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

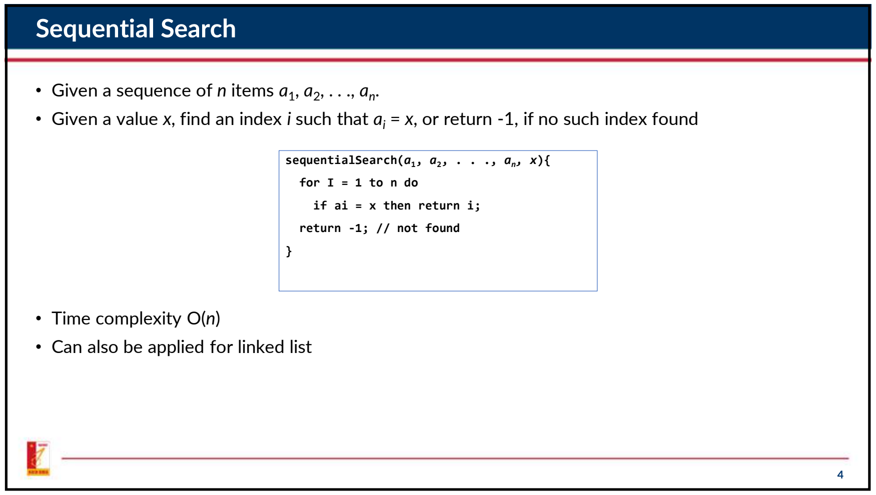
DATA STRUCTURES & ALGORITHMS

SEARCHING

ONE LOVE. ONE FUTURE.


CONTENT

- Sequential search
- Binary search
- Binary Search Trees


Sequential Search

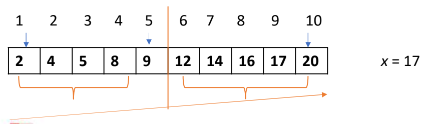
- Given a sequence of *n* items $a_1, a_2, \ldots, a_n$.
- Given a value *x*, find an index *i* such that $a_i = x$, or return -1, if no such index found

```
sequentialSearch(a1, a2, . . ., an, x){
  for I = 1 to n do
    if ai = x then return i;
  return -1; // not found
}
```

- Time complexity O(*n*)
- Can also be applied for linked list

Binary Search

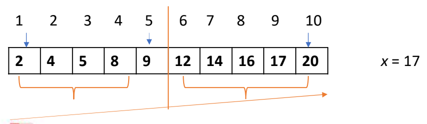
- Sequence of items (indexed from *L*, *L*+1, . . ., *R*) is sorted in a non-increasing (non-decreasing) order of keys
- Base on divide and conquer:
  - Compare the input key with the key of the item in the middle of the sequence and decide to perform binary search on the left subsequence or the right subsequence of the object in the middle



Binary Search

- Sequence of items (indexed from *L*, *L*+1, . . ., *R*) is sorted in a non-increasing (non-decreasing) order of keys
- Base on divide and conquer:
  - Compare the input key with the key of the item in the middle of the sequence and decide to perform binary search on the left subsequence or the right subsequence of the object in the middle
- Running time: O(log(*R*-*L*))

```
binarySearch(a_1, a_2, . . ., a_n, L, R, x){
  if(L = R){
    if(a_L = x) return L;
    return -1;
  }
  m = (L + R)/2;
  if(a_m = x) return m;
  if(a_m < x)
   return binarySearch(a_1, a_2, . . ., a_n,m+1,R,x);
  return binarySearch(a_1, a_2, . . ., a_n,L,m,x);
}
```



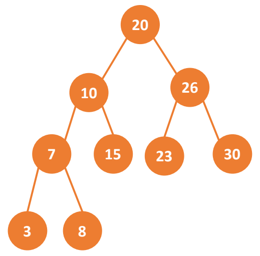
Binary Search

- **Exercise** Given a sequence of distinct elements a_1, a_2, …, a_N and a value *b*. Count the number of pairs (a_i, a_j) having $a_i + a_j = b$ ($i < j$)

Binary Search Trees - BST

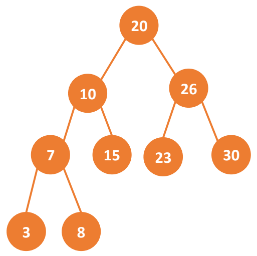
- Binary Search Tree (or BST) is a data structure storing objects under a binary tree:
  - Key of each node is greater than the keys of nodes of the left sub-tree and smaller than the keys of nodes of the right sub-tree

```
struct Node{
  key; // key of the node
  leftChild; // pointer to the left-child
  rightChild; // pointer to the right-child
  parent; // pointer to the parent of the current node
          // parent of the root is null (by convention)
}
```



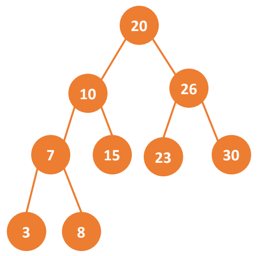
Binary Search Trees - BST

- Binary Search Tree (or BST) is a data structure storing objects under a binary tree:
  - Key of each node is greater than the keys of nodes of the left sub-tree and smaller than the keys of nodes of the right sub-tree
- In-order traversal gets a sequence of keys sorted in an increasing order: 3, 7, 8, 10, 15, 20, 23, 26, 30



Binary Search Trees

- Operations
  - search(*r*, *k*): return the object having key *k* in the BST rooted at *r*
  - insert(*r*, *k*): insert a new object having key *k* to the BST rooted at *r*
  - remove(*r*, *k*): remove the node having key *k* from the BST rooted at *r*
  - findMin(*r*): return the pointer to the object having minimum value of key in the BST rooted at *r*
  - findMax(*r*): return the pointer to the object having maximum value of key in the BST rooted at *r*
  - findSuccessor(*x*): return the pointer to the object having the smallest key but the key is greater than the key of *x*
  - findPredecessor(*x*): return the pointer to the object having the highest key but the key is smaller than the key of *x*



Binary Search Trees

- search(*r*, *k*): return the object having key *k* in the BST rooted at *r*
  - If *k* = *r.key* then return *r*
  - If *k* > *r.key* then perform the search on the right sub-tree
  - If *k* < *r.key* then perform the search on the left sub-tree
- Time complexity O(*h*): *h* is the height of the BST rooted at *r*

```
search(r, k){
  if r = null then return null;
  if k = r.key then return r;
  if k > r.key then
    return search(r.rightChild, k);
  if k < r.key then
    return search(r.leftChild, k);
}
```

Binary Search Trees

- insert(*r*, *k*): insert a new object having key *k* to the BST rooted at *r*, and return pointer to the resulting BST
  - If *k* = *r.key* then return *r* (do not insert)
  - If *k* > *r.key* then perform the insertion on the right sub-tree
  - If *k* < *r.key* then perform the insertion on the left sub-tree
- Time complexity O(*h*): *h* is the height of the BST rooted at *r*

```
insert(r, k){
  if r = null then return Node(k);
  if k = r.key then return r;
  if k > r.key then
    return insert(r.rightChild, k);
  if k < r.key then
    return insert(r.leftChild, k);
}
```

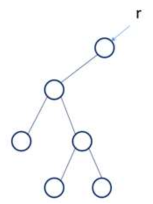
Binary Search Trees

- remove(*r*, *k*): remove the object having key *k* from the BST rooted at *r*, and return pointer to the resulting BST
  - If *k* = *r.key* then perform the removal of the root
  - If *k* > *r.key* then perform the removal of key *k* from the right sub-tree (recursion)
  - If *k* < *r.key* then perform the removal of key *k* from the left sub-tree (recursion)

```
remove(r, k){
  if r = null then return null;
  if k = r.key then
    return removeRoot(r);
  if k > r.key then
    return remove(r.rightChild, k);
  if k < r.key then
    return remove(r.leftChild, k);
}
```

Binary Search Trees

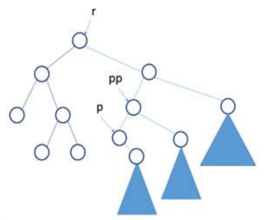
- removeRoot(*r*): remove the the root of the BST rooted at *r*, and return pointer to the resulting BST
  - If *r* does not have right child, then return the pointer to the left child



```
removeRoot(r){
  if r = NULL then return NULL;
  tmp = r;
  if r.rightChild = NULL then {
    r = r.leftChild; free(tmp); return r;
  }
  p = r.rightChild; pp = r;
  if p.leftChild = NULL then {
    r.key = p.key; tmp = p;
    r.rightChild = p.rightChild;
    free(tmp); return r;
  }
  while p.leftChild !=  NULL do {
    pp = p; p = p.leftChild;
  }
  pp.leftChild = p.rightChild; r.key = p.key; free(p);
  return r;
}
```

Binary Search Trees

- removeRoot(*r*): remove the the root of the BST rooted at *r*, and return pointer to the resulting BST
  - If *r* has a right child, then find the node *p* having minimum key of the right sub-tree, then copy the data (key) of *p* to the root *r*, then remove *p* (let the left child pointer of *pp* (parent of *p*) point to the right child of *p*)



```
removeRoot(r){
  if r = NULL then return NULL;
  tmp = r;
  if r.rightChild = NULL then {
    r = r.leftChild; free(tmp); return r;
  }
  p = r.rightChild; pp = r;
  if p.leftChild = NULL then {
    r.key = p.key; tmp = p;
    r.rightChild = p.rightChild;
    free(tmp); return r;
  }
  while p.leftChild !=  NULL do {
    pp = p; p = p.leftChild;
  }
  pp.leftChild = p.rightChild; r.key = p.key; free(p);
  return r;
}
```

Binary Search Trees

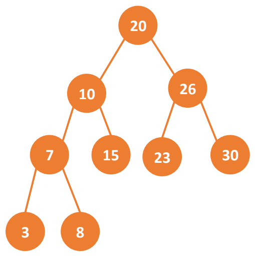
- findMin(*r*): return the pointer to the object (node) having minimum value of key on the BST rooted at *r*.
- findMax(*r*): return the pointer to the object (node) having maximum value of key on the BST rooted at *r*.

```
findMin(r){
  if r = null then return null;
  lmin = findMin(r.leftChild);
  if lmin != null then return lmin;
  return r;
}

findMax(r){
  if r = null then return null;
  rmax = findMax(r.rightChild);
  if rmax != null then return rmax;
  return r;
}
```

Binary Search Trees

- findSuccessor(*x*): return the pointer to the object (node) having the smallest key but the key is greater than the key of *x*
  - Successor(20) = 23
  - Successor(30) = null
  - Successor(8) = 10



Binary Search Trees

- findSuccessor(*x*): return the pointer to the object (node) having the smallest key but the key is greater than the key of *x*
  - If *x* has a right child, then the successor of *x* is the object *y* with *y.key* is minimum among objects of the right sub-tree of *x*: function findMin(*x.rightChild*)

```
findSuccessor(x){
  if x = null return null;
  if r.rightChild != null then
    return findMin(r.rightChild);

  p = x.parent;
  while p != null do {
    if p.leftChild != null then return p;
    p = p.parent;
  }
  return null; // not found successor
}
```

Binary Search Trees

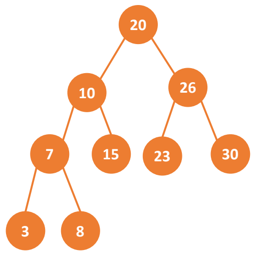
- findSuccessor(*x*): return the pointer to the object (node) having the smallest key but the key is greater than the key of *x*
  - If *x* has a right child, then the successor of *x* is the object *y* with *y.key* is minimum among objects of the right sub-tree of *x*: function findMin(*x.rightChild*)
  - If *x* does not have a right child, then the successor of x is the nearest ancestor of x (use the *parent* pointer of nodes) having a left child (which is *x* or an ancestor of *x*).

```
findSuccessor(x){
  if x = null return null;
  if r.rightChild != null then
    return findMin(r.rightChild);

  p = x.parent;
  while p != null do {
    if p.leftChild != null then return p;
    p = p.parent;
  }
  return null; // not found successor
}
```

Binary Search Trees

- findPredecessor(*x*): return the pointer to the object having the highest key but the key is smaller than the key of *x*
  - Predecessor(20): 15
  - Predecessor(3) = null
  - Predecessor(23) = 20



Binary Search Trees

- findPredecessor(*x*): return the pointer to the object having the highest key but the key is smaller than the key of *x*
  - If *x* has a left child, then the predecessor of *x* is the object *y* with *y.key* is maximum among objects of the left sub-tree of *x*: function findMax(*x.leftChild*)

```
findPredecessor(x){
  if x = null return null;
  if r.leftChild != null then
    return findMax(r.leftChild);

  p = x.parent;
  while p != null do {
    if p.rightChild != null then return p;
    p = p.parent;
  }
  return null; // not found predecessor
}
```

Binary Search Trees

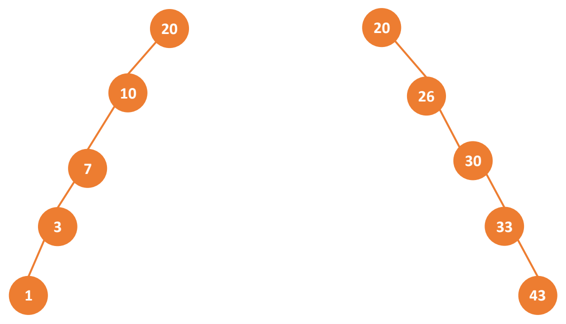
- findPredecessor(*x*): return the pointer to the object having the highest key but the key is smaller than the key of *x*
  - If *x* has a left child, then the predecessor of *x* is the object *y* with *y.key* is maximum among objects of the left sub-tree of *x*: function findMax(*x.leftChild*)
  - If *x* does not have a left child, then the predecessor of x is the nearest ancestor of x (use the *parent* pointer of nodes) having a right child (which is *x* or an ancestor of *x*).

```
findPredecessor(x){
  if x = null return null;
  if r.leftChild != null then
    return findMax(r.leftChild);

  p = x.parent;
  while p != null do {
    if p.rightChild != null then return p;
    p = p.parent;
  }
  return null; // not found predecessor
}
```

Binary Search Trees

- Time complexity of most of the basic operations on a BST is proportional to the height of the BST.
- Worst case: O(*n*) (in which n is the number of nodes of the BST)



HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !