Lesson 5. Data Structures (1): Linked List and Binary Search Tree

I111E – Algorithms and Data Structures

Ryuhei Uehara & Giovanni Viglietta
uehara@jaist.ac.jp & johnny@jaist.ac.jp

JAIST – October 30, 2019

All material is available at
www.jaist.ac.jp/~uehara/couse/2019/i111e

Goals of today's lecture

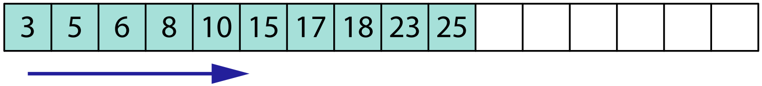
- Learn about Linked Lists
  - Searching for data in a Linked List
  - Inserting data in a Linked List
  - Deleting data from a Linked List
- Learn about Binary Search Trees
  - Searching for data in a Binary Search Tree
  - Inserting data in a Binary Search Tree
  - Deleting data from a Binary Search Tree

Algorithms and data structures

- **Algorithm:** how to solve a problem
- **Data structure:** how to organize data
  - Format of the intermediate results of a computation
  - Contributes to the efficiency of algorithms
  - Examples: Array, Linked List, Stack, Queue, Tree, ...

Array

- Data are organized in a sequence
- Accessing any element takes constant time (RAM model)
  - Other structures may be accessed only from specific points (e.g., Linked Lists: accessing the ith element takes $O(i)$ time)
- Can be accessed in order of indices (i.e., sequentially)
  - Other structures may lack this property (e.g., Tree structures)



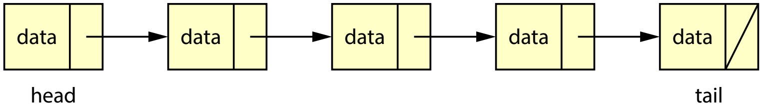
Linked List

In a Linked List, data are organized in nodes. Each node contains:

- Some data,
- A pointer to the next node.

Typically, Linked Lists are used to organize data in a chain:

- The first node is the head, and is not pointed to by any node.
- The last node is the tail, and points to NULL.



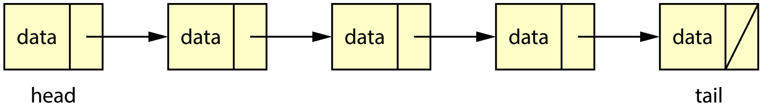
Linked List

In a Linked List, data are organized in nodes. Each node contains:

- Some data,
- A pointer to the next node.

Typically, Linked Lists are used to organize data in a chain:

- The first node is the head, and is not pointed to by any node.
- The last node is the tail, and points to NULL.



- Other variants include Two-Way Linked Lists, where each node also points to the previous node.
- Linked Lists can also be used to represent Tree structures (where each node points to its parent).

Linked List: implementation of a node

This is a C implementation of a (One-Way) Linked List node:

```
typedef struct list_node {
    int value;
    struct list_node* next;
} list_node;
```

We can create a Linked List as follows:

```
list_node* head = malloc(sizeof(list_node));
list_node* middle = malloc(sizeof(list_node));
list_node* tail = malloc(sizeof(list_node));
head -> value = 10;
head -> next = middle;
middle -> value = 20;
middle -> next = tail;
tail -> value = 30;
tail -> next = NULL;
```

Searching a Linked List

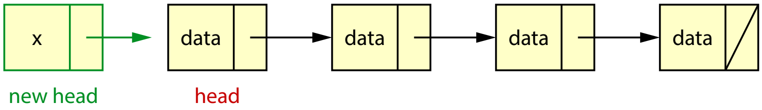
To search a Linked List for x, we scan its nodes one by one, starting from the head, and stopping when we find x or the next pointer is NULL:

```
list_node* list_search(list_node* head, int x) {
    list_node* node = head;
    while (node != NULL) {
        if (node -> value == x) return node;
        node = node -> next;
    }
    return NULL;
}
```

If there are n nodes in the Linked List, the search takes $O(n)$ time.

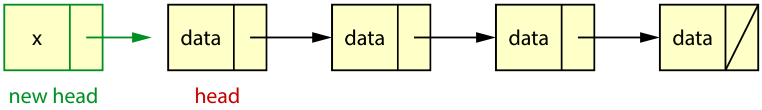
Inserting data in a Linked List

To insert a new value x in a Linked List, we can store x in a new node, and make it point to the head of the Linked List:



Inserting data in a Linked List

To insert a new value x in a Linked List, we can store x in a new node, and make it point to the head of the Linked List:

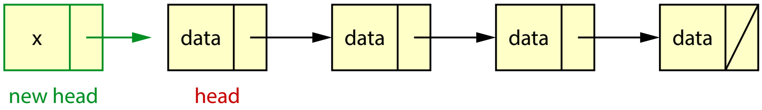



```
list_node* list_insert(list_node* head, int x) {
    list_node* node = malloc(sizeof(list_node));
    node -> value = x;
    node -> next = head;
    return node;
}
```

The insertion function takes constant time.

Inserting data in a Linked List

To insert a new value x in a Linked List, we can store x in a new node, and make it point to the head of the Linked List:




```
list_node* list_insert(list_node* head, int x) {
    list_node* node = malloc(sizeof(list_node));
    node -> value = x;
    node -> next = head;
    return node;
}
```

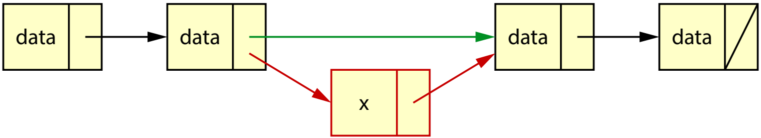
The insertion function takes constant time.

Note: this insertion method does not keep the Linked List sorted. To keep it sorted, we would have to scan it an insert every new element at the right position: this variant takes $O(n)$ time.

Deleting data from a Linked List

To delete a value x from a Linked List:

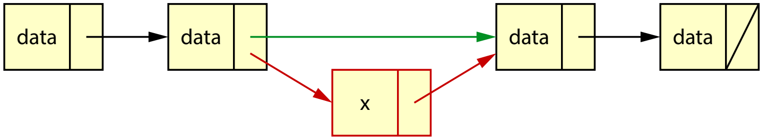
- Search for the node containing x,
- Delete it,
- Make the previous node point to the next node.



Deleting data from a Linked List

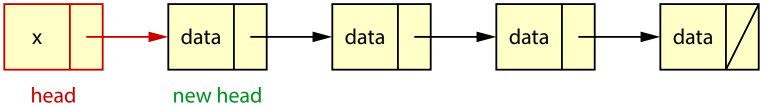
To delete a value x from a Linked List:

- Search for the node containing x,
- Delete it,
- Make the previous node point to the next node.



Special case:

- If x is in the head node, the second node becomes the head.



Deleting data from a Linked List

This is an implementation of the deletion algorithm:

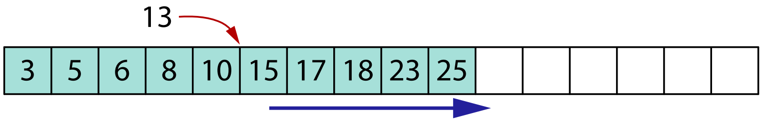
```
list_node* list_delete(list_node* head, int x) {
   list_node* previous = NULL;
   list_node* current = head;
   while (current != NULL && current -> value != x) {
      previous = current;
      current = current -> next;
   }
   if (current == NULL) return head;
   if (previous == NULL) {
      list_node* n = current -> next;
      free(current);
      return n;
   }
   previous -> next = current -> next;
   free(current);
   return head;
}
```

The worst-case running time is $O(n)$.

Arrays vs. Linked Lists

Arrays:

- Every element is easy to access: $O(1)$.
- Inserting and deleting elements is complicated (involves shifting and possibly re-allocating the entire Array).
- If the Array is sorted, binary search takes $O(\log n)$ time.

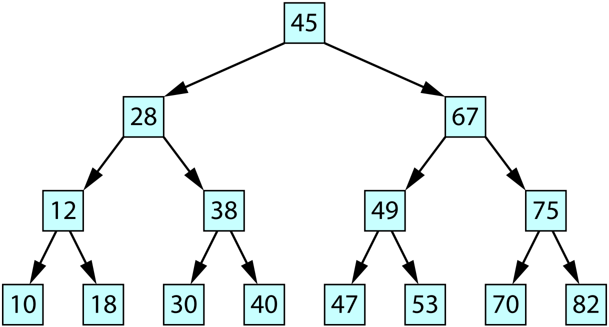


Linked Lists:

- To access the ith element, we have to reach it: $O(i)$.
- Inserting and deleting elements is easy.
- Even if the Linked List is sorted, searching takes $O(n)$ time.

Binary Search Tree

A Binary Search Tree is the natural data structure on which to perform binary search:



The key property of a BST is that, for each node v,

- Its left subtree contains all nodes with lower value than v,
- Its right subtree contains all nodes with greater value than v.

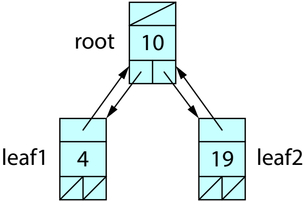
Binary Search Tree node

In our implementation of a BST node, we have a pointer to each child of the node, and also a pointer to its parent.

```
typedef struct tree_node {
    int value;
    struct tree_node* parent;
    struct tree_node* left;
    struct tree_node* right;
} tree_node;
```

We can set up a BST as follows:

```
tree_node* root = malloc(sizeof(tree_node));
tree_node* leaf1 = malloc(sizeof(tree_node));
tree_node* leaf2 = malloc(sizeof(tree_node));
root -> value = 10;
root -> parent = NULL;
root -> left = leaf1;
root -> right = leaf2;
leaf1 -> value = 4;
leaf1 -> parent = root;
leaf1 -> left = NULL;
leaf1 -> right = NULL;
leaf1 -> value = 19;
leaf2 -> parent = root;
leaf2 -> left = NULL;
leaf2 -> right = NULL;
```



Searching a Binary Search Tree

Searching a BST is done like with a Linked List,

but we choose the left or right child of each node we visit

based on the value stored in the node:

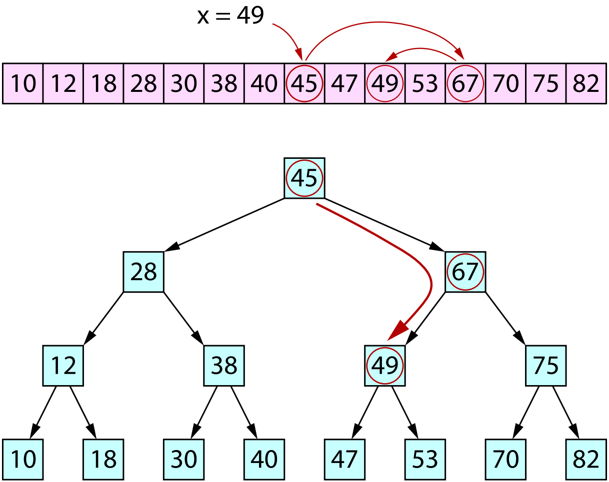
```
tree_node* tree_search(tree_node* root, int x) {
    tree_node* node = root;
    while (node != NULL) {
        if (node -> value == x) return node;
        if (node -> value > x) node = node -> left;
        else node = node -> right;
    }
    return NULL;
}
```

The running time of searching depends on the height of the BST:

- If the BST is balanced, searching takes $O(\log n)$ time.
- If the BST is not balanced, searching takes up to $O(n)$ time.

Searching a Binary Search Tree

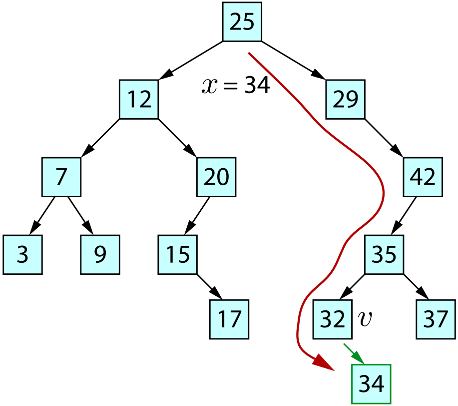
Searching a BST corresponds to doing binary search on an Array:
Each path in a BST is a sequence of comparisons in a sorted Array.



Inserting data in a Binary Search Tree

To insert a value x in a BST:

- Search for x in the BST: the search ends in a node v.
- Create a new node and store x in it,
- Attach the new node as a left or right child of v.



Inserting data in a Binary Search Tree

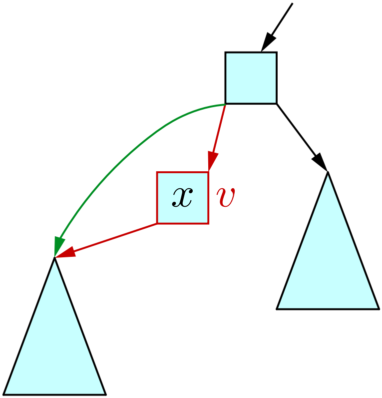
This is an implementation of the insertion algorithm:

```
tree_node* tree_insert(tree_node* root, int x) {
   tree_node* previous = NULL;
   tree_node* current = root;
   while (current != NULL) {
      previous = current;
      if (current -> value == x) return true;
      if (current -> value > x) current = current -> left;
      else current = current -> right;
   }
   tree_node* node = malloc(sizeof(tree_node));
   node -> value = x;
   node -> parent = previous;
   node -> left = NULL;
   node -> right = NULL;
   if (previous == NULL) return node;
   if (previous -> value > x) previous -> left = node;
   else previous -> right = node;
   return root;
}
```

Deleting data from a Binary Search Tree

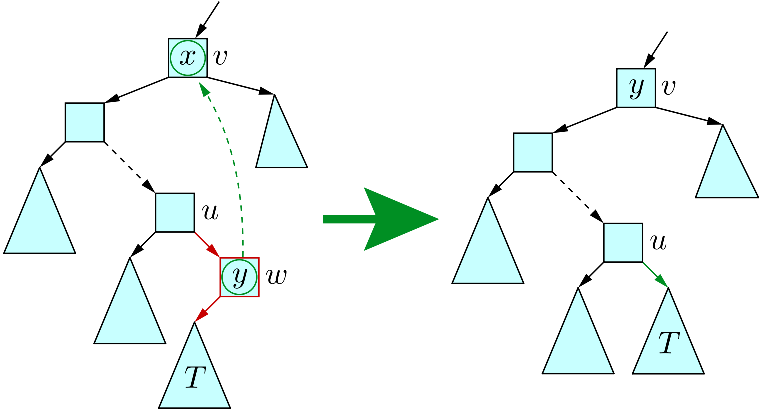
To delete a value x from a BST:

- Search for x in the BST: the search ends in a node v.
- **Case 1:** If one of the two children of v is empty:
  - Attach the other child of v to the parent of v,
  - Delete v.



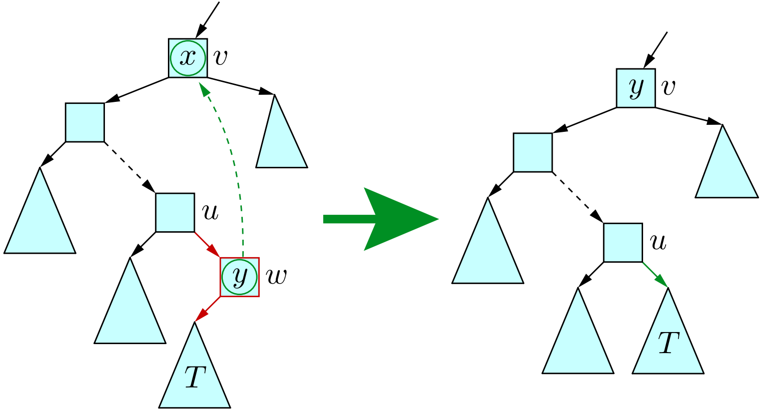
Deleting data from a Binary Search Tree

- **Case 2:** If both children of v are non-empty:
  - Find the node w with largest value in the left subtree of v,
  - Copy the value of w into v,
  - Remove w as in Case 1 (note: the right child of w is empty).



Deleting data from a Binary Search Tree

- **Case 2:** If both children of v are non-empty:
  - Find the node w with largest value in the left subtree of v,
  - Copy the value of w into v,
  - Remove w as in Case 1 (note: the right child of w is empty).



Exercise: prove that the resulting structure is still a BST.

Deleting data from a Binary Search Tree

To implement the deletion algorithm, we use a helper function:

```
void transplant(tree_node* to, tree_node* from) {
    tree_node* p = to -> parent;
    if (from != NULL) from -> parent = p;
    if (p == NULL) return;
    if (p -> value > to -> value) p -> left = from;
    else p -> right = from;
}
```

This function takes a subtree rooted at node `from`

and attaches it in place of node `to`.

We will use it to remove a node and attach its child to its parent.

Deleting data from a Binary Search Tree

This is an implementation of the deletion algorithm:

```
tree_node* tree_delete(tree_node* head, int x) {
    tree_node* v = tree_search(head, x);
    if (v == NULL) return head;
    if (v -> left != NULL && v -> right != NULL) {
        tree_node* w = v -> left;
        while (w -> right != NULL) w = w -> right;
        transplant(w, w -> left);
        v -> value = w -> value;
        free(w);
        return head;
    }
    tree_node* u;
    if (v -> left == NULL) u = v -> right;
    else u = v -> left;
    transplant(v, u);
    tree_node* p = v -> parent;
    free(v);
    if (p == NULL) return u;
    return head;
}
```

Performance of Binary Search Trees

The performances of search, insertion, and deletion in a BST are the same, and depend on how balanced the tree is:

- If the BST is well balanced, they have the same performance as binary search: $O(\log n)$ in the worst case.
- If the BST is very unbalanced, it looks like a Linked List, and its performance is $O(n)$ in the worst case.

Performance of Binary Search Trees

The performances of search, insertion, and deletion in a BST are the same, and depend on how balanced the tree is:

- If the BST is well balanced, they have the same performance as binary search: $O(\log n)$ in the worst case.
- If the BST is very unbalanced, it looks like a Linked List, and its performance is $O(n)$ in the worst case.

The shape of a BST depends on the initial data
and on the order of insertion and deletions:
on average, we should expect a BST to remain fairly balanced.

However, there are also self-balancing versions of the BST, whose insertion and deletion operations maintain it well balanced (e.g., AVL trees, red-black trees, B-trees, ...).