ECE 250 Algorithms and Data Structure
Project Three: Leftist heap

Soheil Soltani

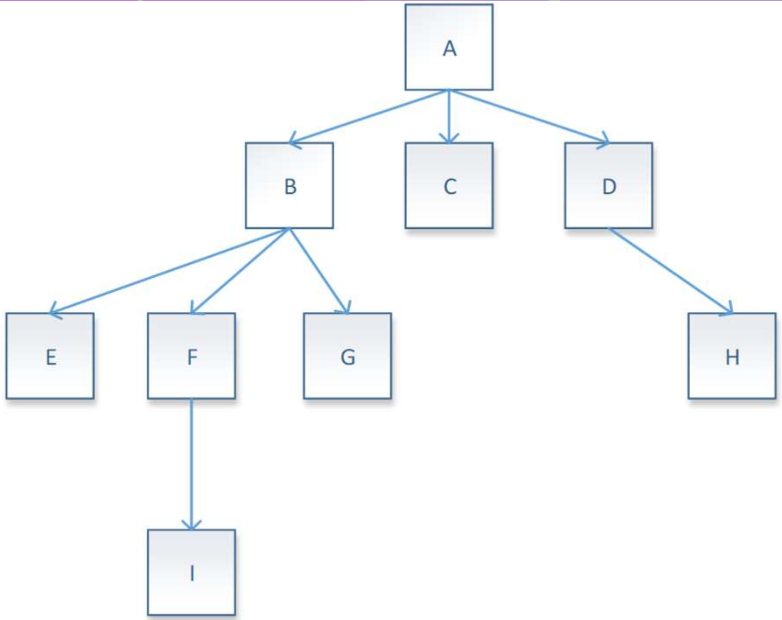
October 18, 2018

UNIVERSITY OF WATERLOO

Tree Data Structure

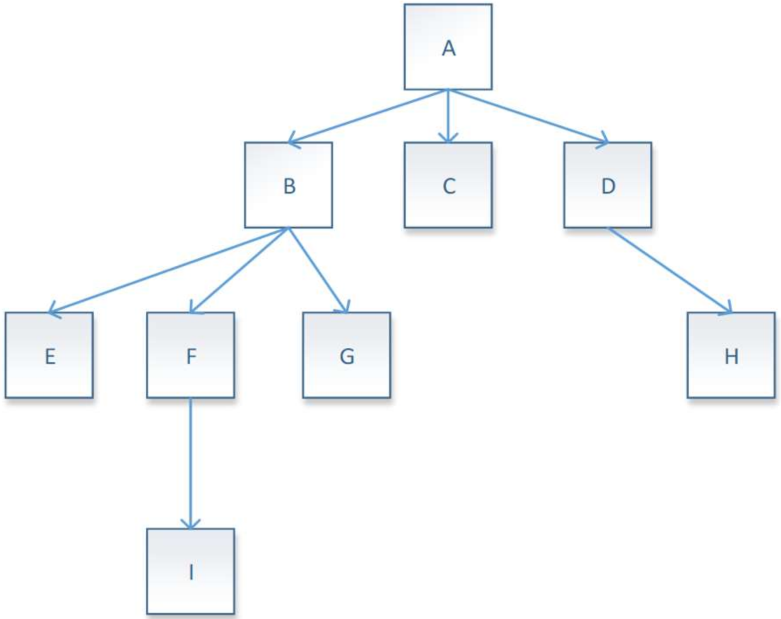
- Several Data Structures:
  - Arrays:
    - Random Access
    - Easy to implement
  - Linked lists:
    - Ideal for frequent add, remove, and update
  - Drawbacks:
    - Time needed to search for an item:
      - Both arrays and linked lists are ***linear*** structures. The time needed to search a ***linear*** list is proportional to the size.
      - Imagine that you want to search for ***m*** items in an array with 1,000,000 elements. Even on a machine with 1 million comparison per second, this search takes m seconds which is not acceptable.
  - Solution:
    - Find more efficient data structures such as tree.

Tree Data Structure



- Tree:
  - A collection of nodes connected by ***edges***
  - Nonlinear (Hierarchical) compared to arrays, linked lists, stacks, and queues
  - Abstract Data Types (ADT)
  - Can be empty
  - One node is distinguished as a ***root***
  - Every node (except root) is connected by a directed ***edge*** from ***exactly*** one other node

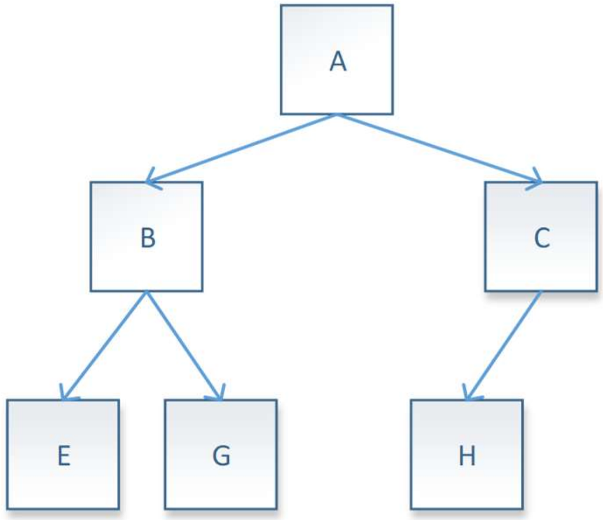
Tree Data Structure



- Tree:
  - A is the ***parent*** of B,C, and D
  - B is a ***child*** of A
  - B is the parent of E, F, and G
  - B, C, and D are ***siblings***
  - Nodes with no children are called ***leaves*** or ***external*** nodes
  - Nodes that are not leaves are called ***internal*** nodes
  - The ***depth*** of a node is the number of edges from root to that node
  - The ***height*** of a node is the number of edges from the node to the deepest leaf
  - The ***height*** of a tree is the height of its root

Tree Data Structure

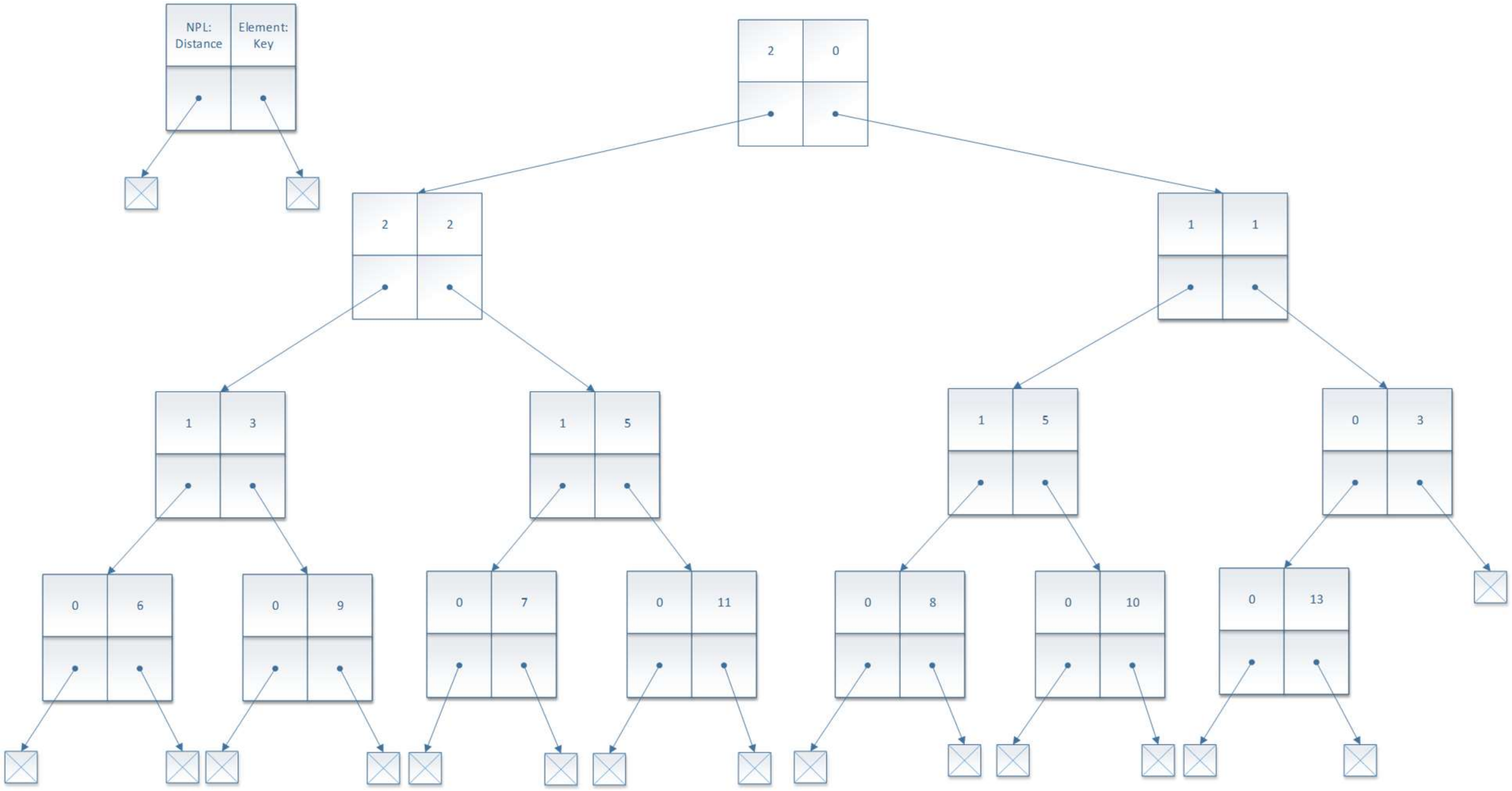
- Binary Tree:
  - A tree where each node can have no more than ***two*** children:
    - The ***left*** child
    - The ***right*** child
  - E is the left child of B
  - G is the right child of B



Leftist Heap

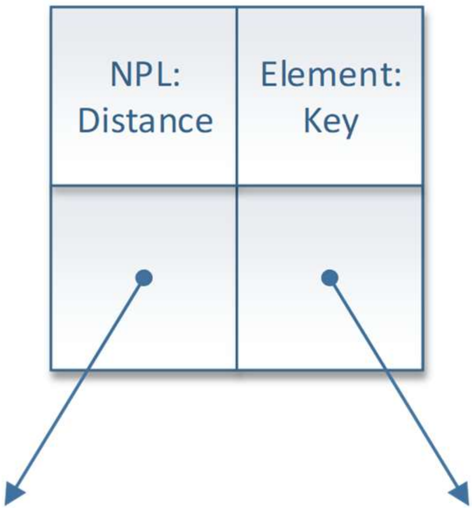
- Leftist tree: is an implementation of a ***mergeable heap***
  - Mergeable heap: is a ***heap*** that supports an additional operation (i.e., merging two heaps)
    - Heap: is a ***tree-based*** ADT that satisfies ***heap property***
      - Heap property (Max heap): the key (value) of each parent node is greater than or equal to its children
      - Heap property (Min heap): the key (value) of each parent node is less than or equal to its children
- Usage: priority queues
- ***Distance*** (null_path_length): the number of edges in the shortest path from a node to its descendent external nodes
- Properties of leftist tree:
  - Key(i)≥key(parent (i))
  - Distance(right(i)) ≤ Distance(left(i))
- Note:
  - Each sub-tree of a leftist tree is also a leftist tree
  - Distance(i)=1+Distance(right(i))

Leftist Heap Example (i.e., int.in.txt)



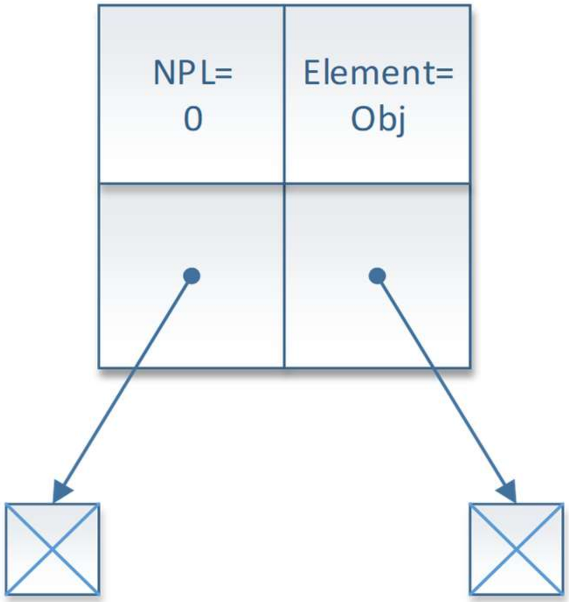
Leftist_node

```
template <typename Type>
class Leftist_node {
private:
Type element;
Leftist_node *left_tree;
Leftist_node *right_tree;
int heap_null_path_length;
public:
Leftist_node(Type const &);
Type retrieve() const;
bool empty() const;
Leftist_node *left() const;
Leftist_node *right() const;
int count(Type const &) const;
int null_path_length() const;
void push(Leftist_node *, Leftist_node *&);
void clear(); };
```

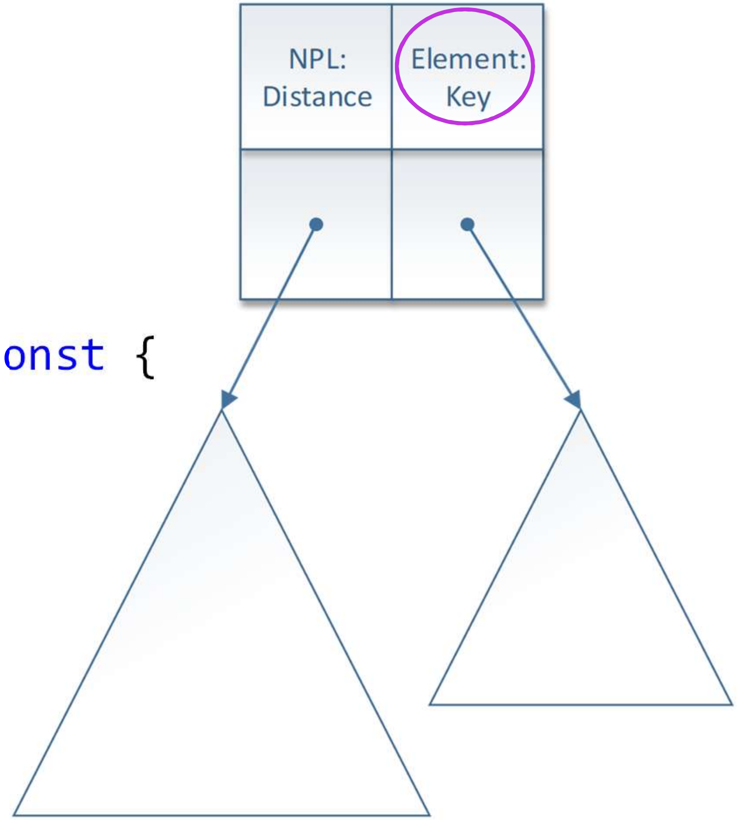


Leftist_node Constructor

```
template <typename Type>
Leftist_node<Type>::Leftist_node(Type const &obj) :
element(obj),
left_tree(nullptr),
right_tree(nullptr),
heap_null_path_length(0) {
// does nothing
}
```

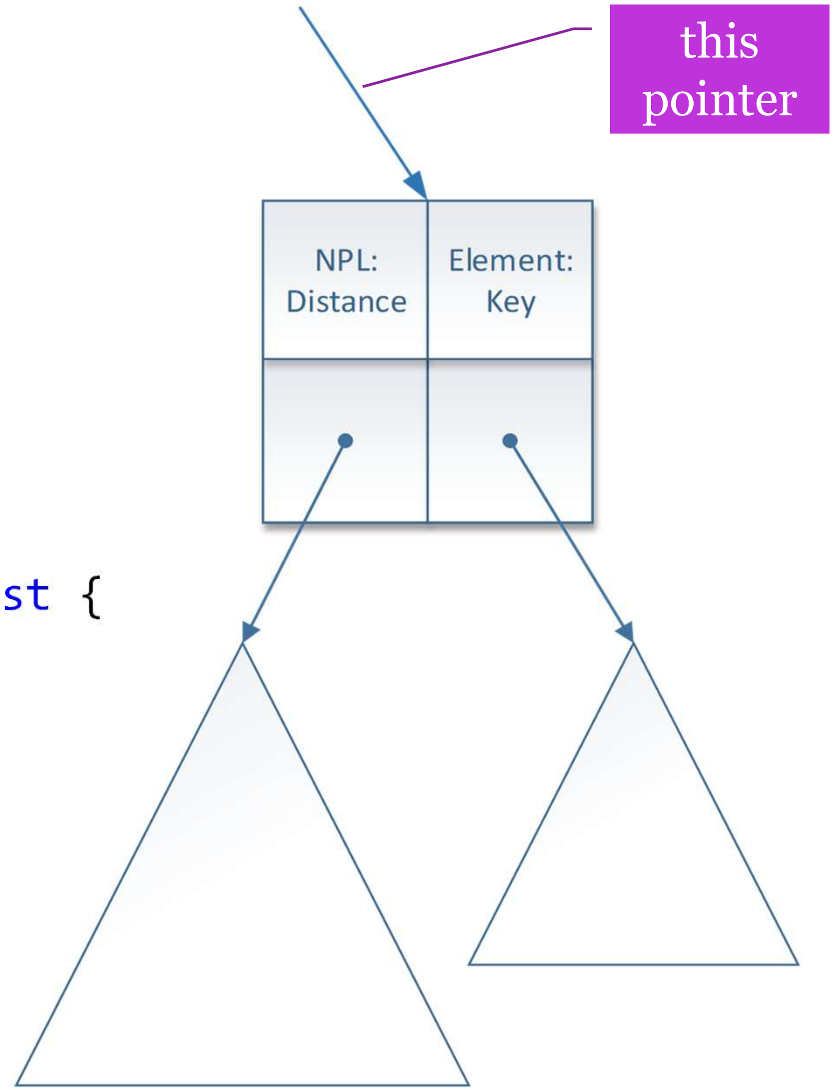


Leftist_node:: retrieve()




```
template <typename Type>
Type Leftist_node<Type>::retrieve() const {
//Return the element.
}
```

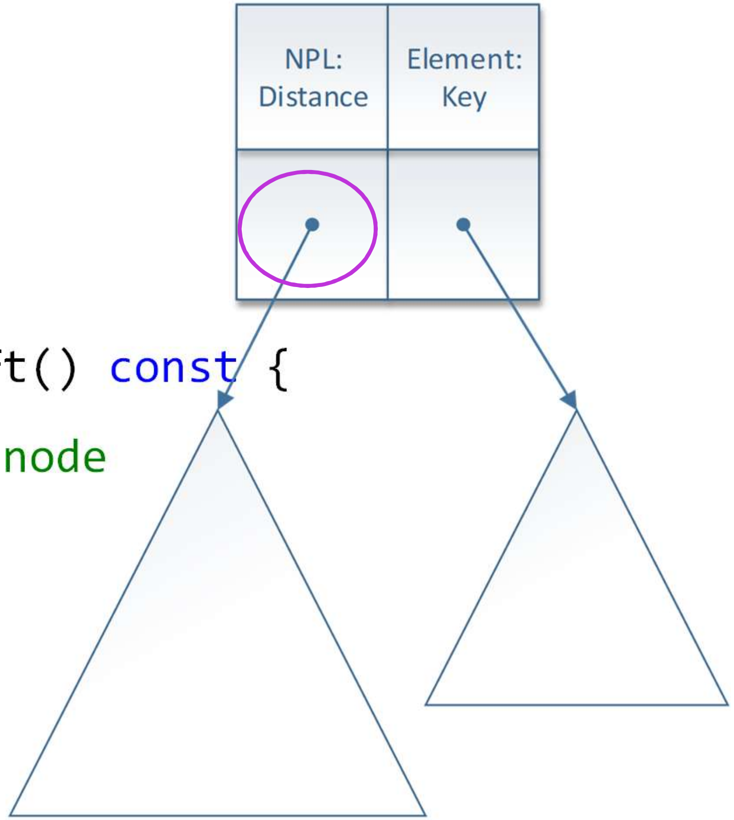
Leftist_node:: empty()



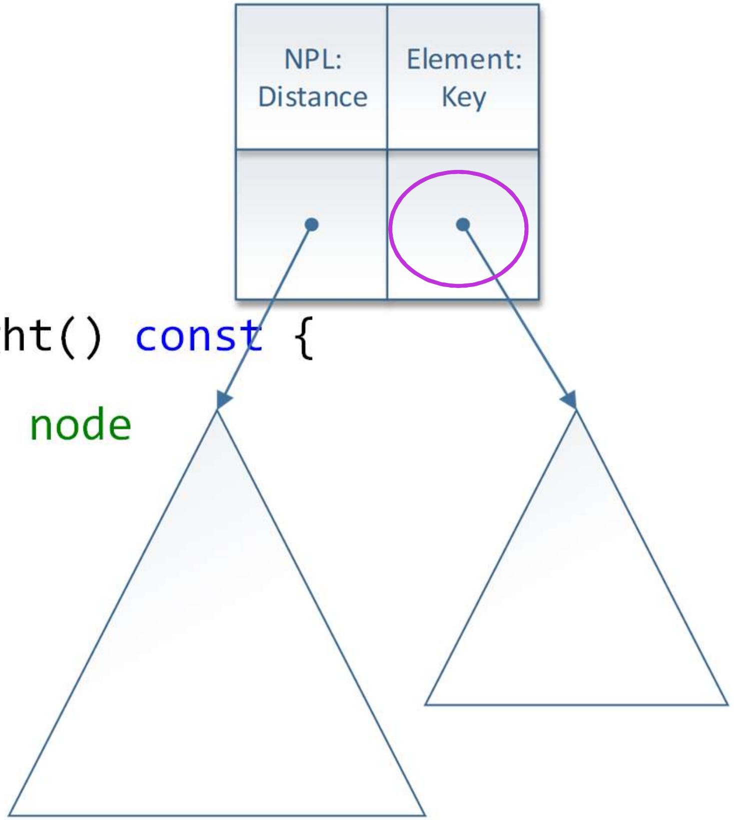

```
template <typename Type>
bool Leftist_node<Type>::empty() const {
//Check if this equals to nullptr.


}
```

Leftist_node:: left()

```
template <class Type>
Leftist_node<Type> *Leftist_node<Type>::left() const {
// Return the address of left tree of this node
}
```

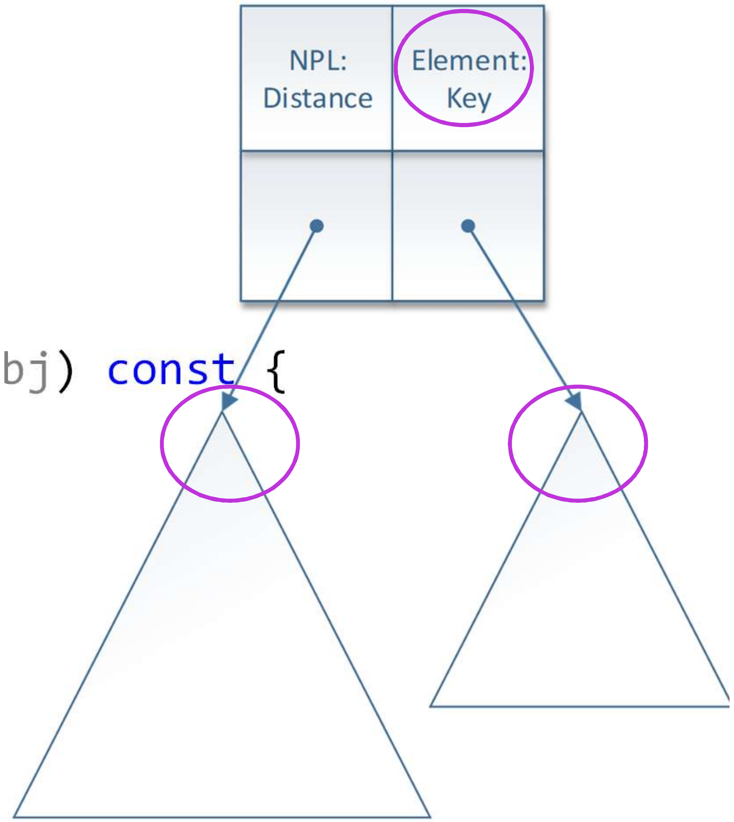


Leftist_node:: right()



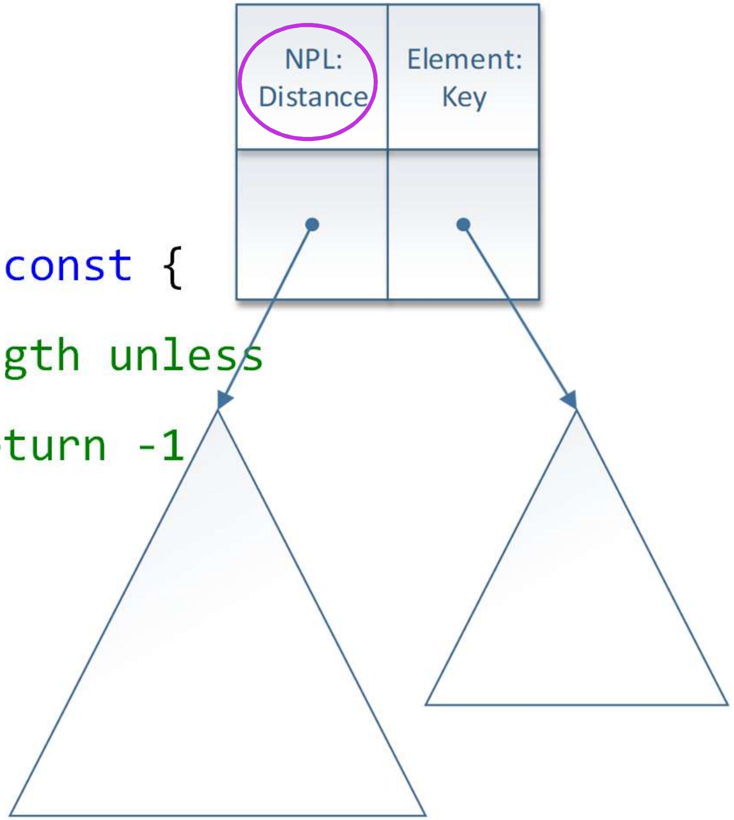

```
template <class Type>
Leftist_node<Type> *Leftist_node<Type>::right() const {
// Return the address of right tree of this node
}
```

Leftist_node:: count()




```
template <typename Type>
int Leftist_node<Type>::count(Type const &obj) const {
// Return the number of instances of obj in
this sub-tree.
// You can do it recursively
}
```

Leftist_node:: null_path_length()



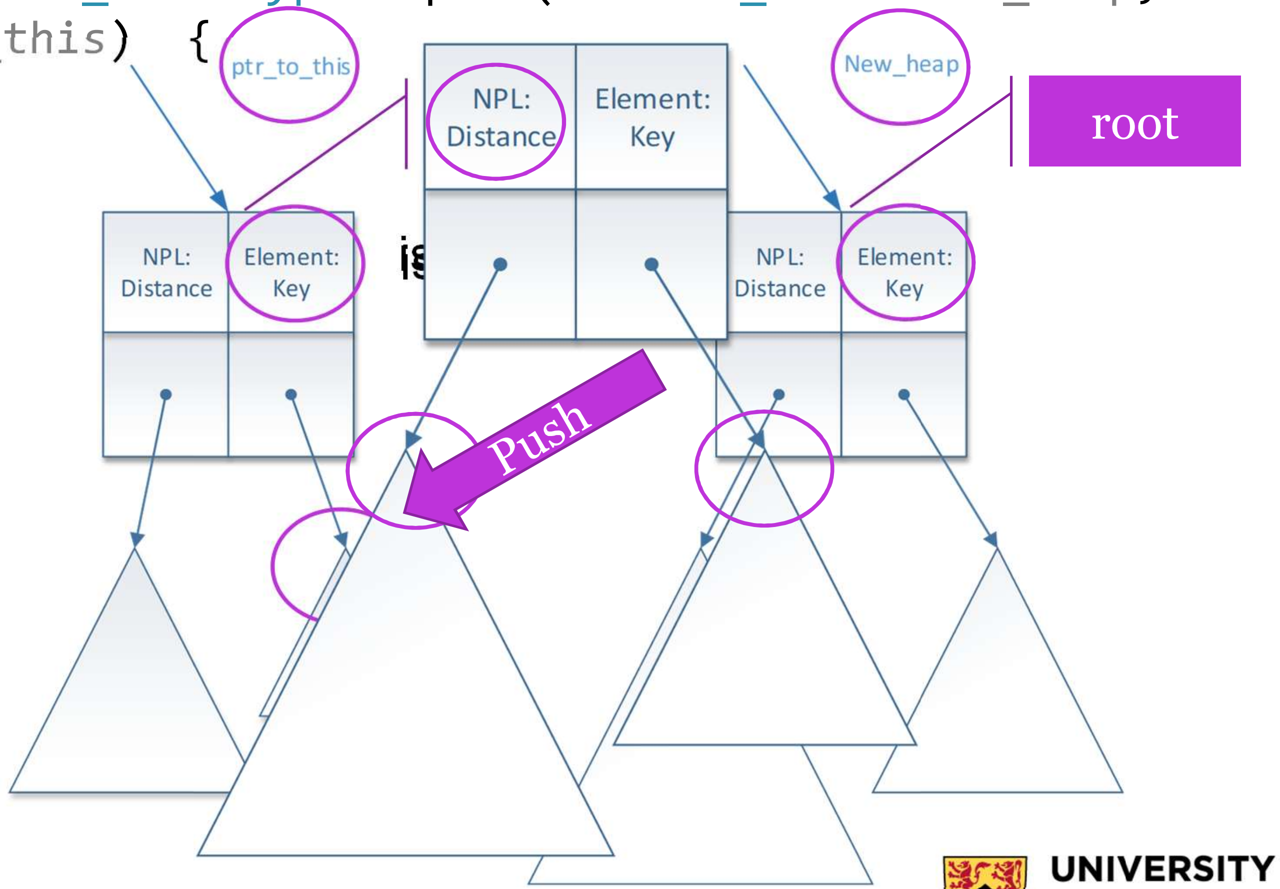

```
template <typename Type>
int Leftist_node<Type>::null_path_length() const {
// Return the member variable null-path length unless
 this isthe null pointer, in which case, return -1
}
```

Leftist_node:: push

```
template <typename Type>

void Leftist_node<Type>::push(Leftist_node *new_heap, Leftist_node *&ptr_to_this)  {

//If the new_heap is null return

//if this is null, set the pointer to this to be the new heap and return

// If the element (value) of current node > new_heap’s element, set the pointer to
this to be the new heap and push this node to the new heap

//If the element of this node ≤  new_heap’s element, push the node into the right
subree.

// Update the null_path length

// if the left sub-tree has a smaller null_path_length than the right sub-tree, swap
the two sub-trees

}
```

Leftist_node:: push

```
template <typename Type>

void Leftist_node<Type>::push(Leftist_node *new_heap, Leftist_node
*&ptr_to_this)  {

}
```

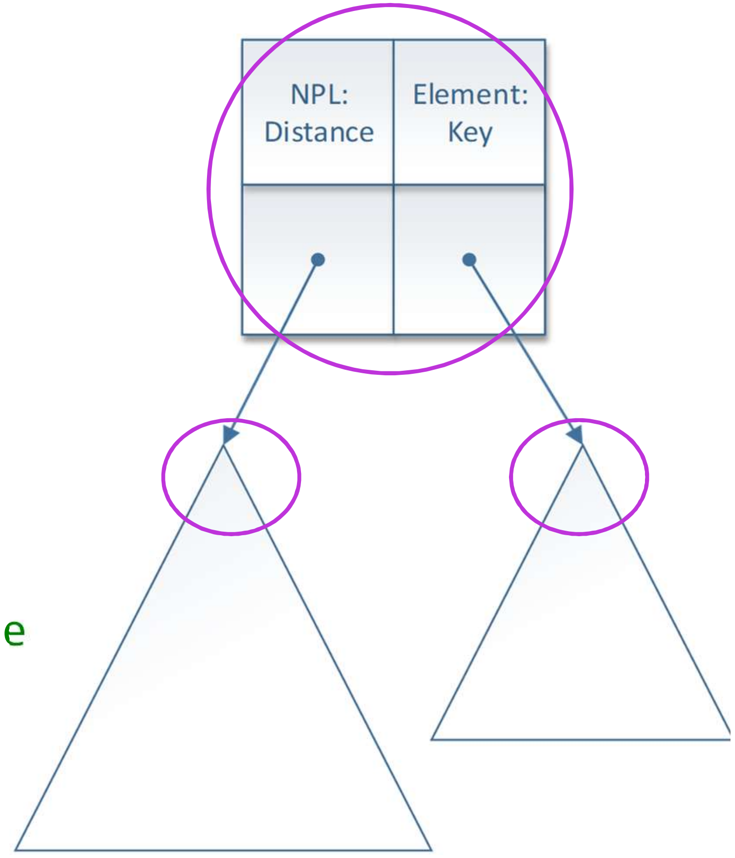


Why *&ptr_to_this?

- Why don't we use "this pointer"?
  - "this pointer" is a prvalue
  - Similar to e.g., 55 : 55=x ?!!
  - You can not reassign "this pointer"
- So we need to pass the address of current node:
  - Similar to passing by reference or passing by value in C++:
    - Passing by reference: the caller and the callee use the same variable
    - Passing by value: the caller and the callee have two different variables

Leftist_node::clear()

```
template<typename Type>
void Leftist_node<Type>::clear() {
//If new heap is null return
// Call clear function on the left sub-tree
// Call clear function on the right sub-tree
// Delete this node
}
```



Leftist_heap

```
class Leftist_heap {
private:
    Leftist_node<Type> *root_node;
    int heap_size;
public:
    Leftist_heap();
    ~Leftist_heap();
    void swap(Leftist_heap &heap);
    bool empty() const;
    int size() const;
    int null_path_length() const;
    Type top() const;
    int count(Type const &) const;
    void push(Type const &);
    Type pop();
    void clear();
// Friends
template <typename T>
friend std::ostream &operator<<(std::ostream &, Leftist_heap<T> const &);
};
```


UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Leftist_heap Constructor

```
template <typename Type>
Leftist_heap<Type>::Leftist_heap() :
root_node(nullptr),
heap_size(0) {
// does nothing
}
```

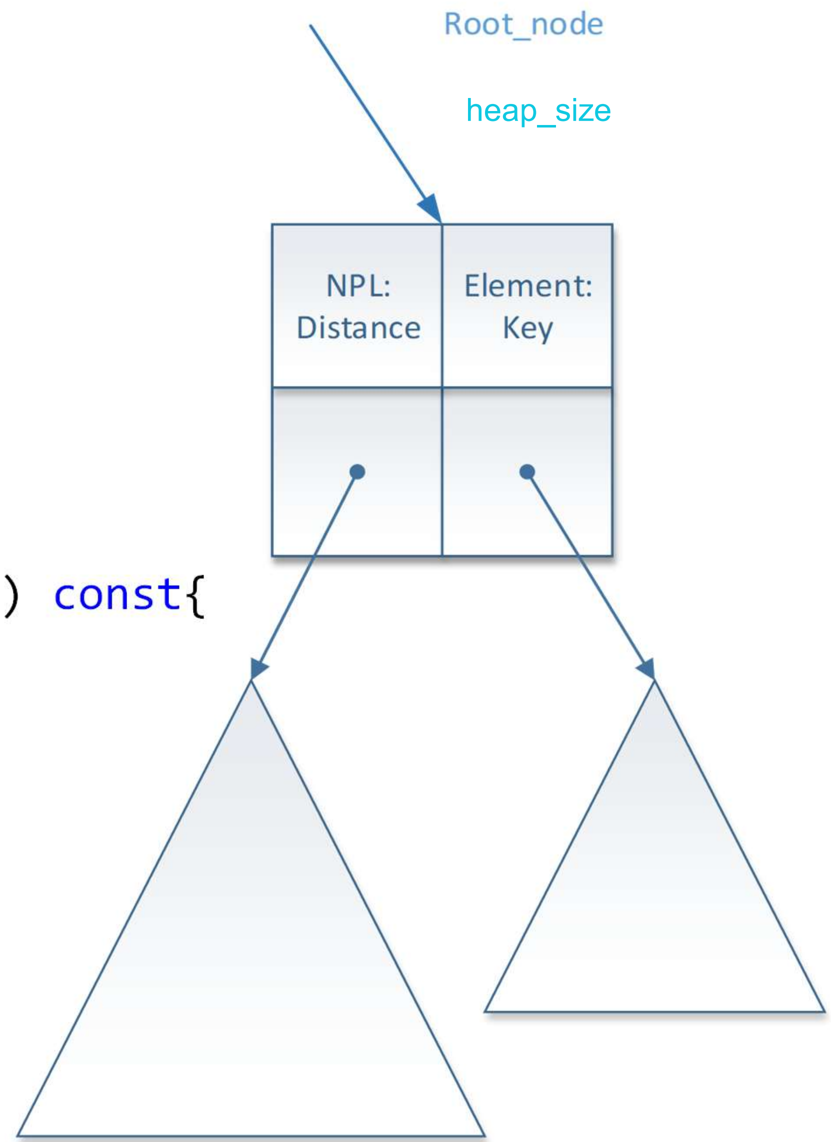
Leftist_heap destructor

```
template <typename Type>
Leftist_heap<Type>::~Leftist_heap() {
clear();// might as well use it...
}
```

Leftist_heap::swap()

```
template <typename Type>
void Leftist_heap<Type>::swap(Leftist_heap<Type> &heap) {
std::swap(root_node, heap.root_node);
std::swap(heap_size, heap.heap_size);
}
```

Leftist_heap::empty()




```
template <typename Type>
bool Leftist_heap<Type>::empty() const{
// Check if the heap is empty
}
```

Leftist_heap::size()

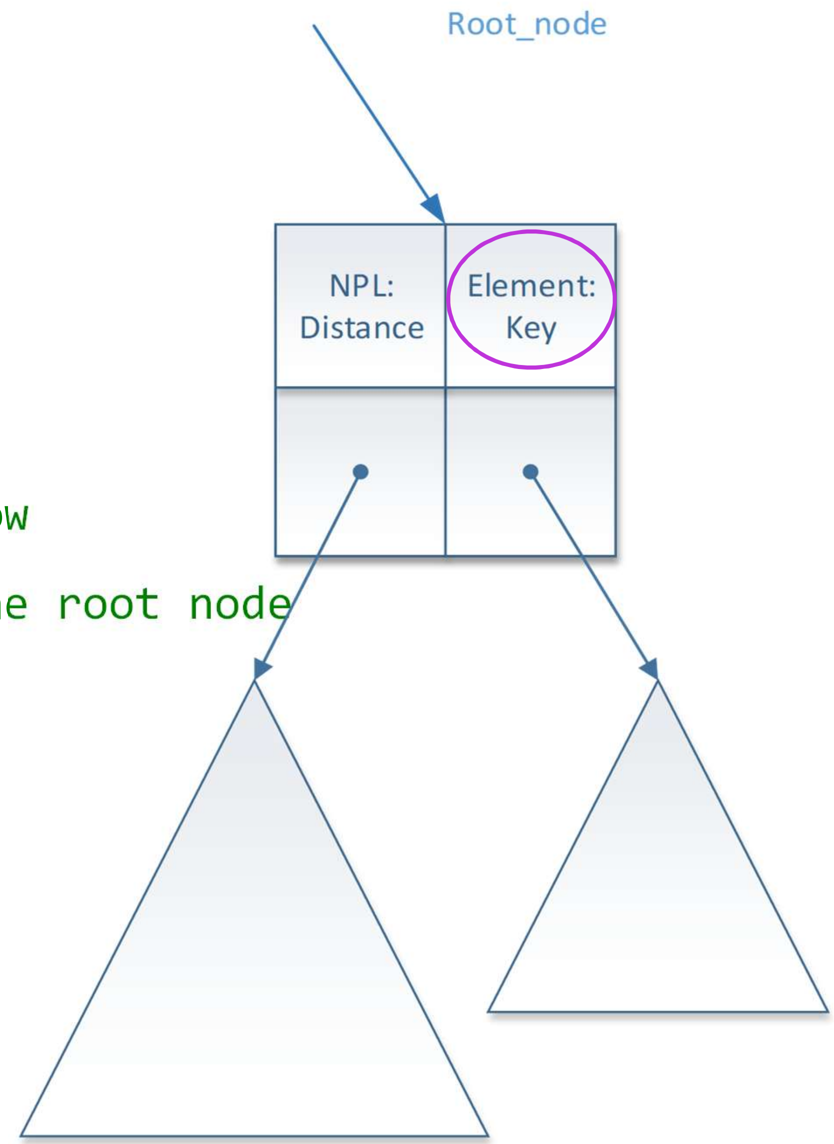
```
template<typename Type>
int Leftist_heap<Type>::size() const{
// Return the number of nodes in the heap
}
```

Leftist_heap::null_path_length()

```
template<typename Type>
int Leftist_heap<Type>::null_path_length() const {
// Return the heap_null_path_length of the root node
}
```

Leftist_heap::top()

```
template<typename Type>
Type Leftist_heap<Type>::top() const{
// If the heap is empty throw underflow
// Otherwise, return the element of the root node
}
```



Leftist_heap::count()

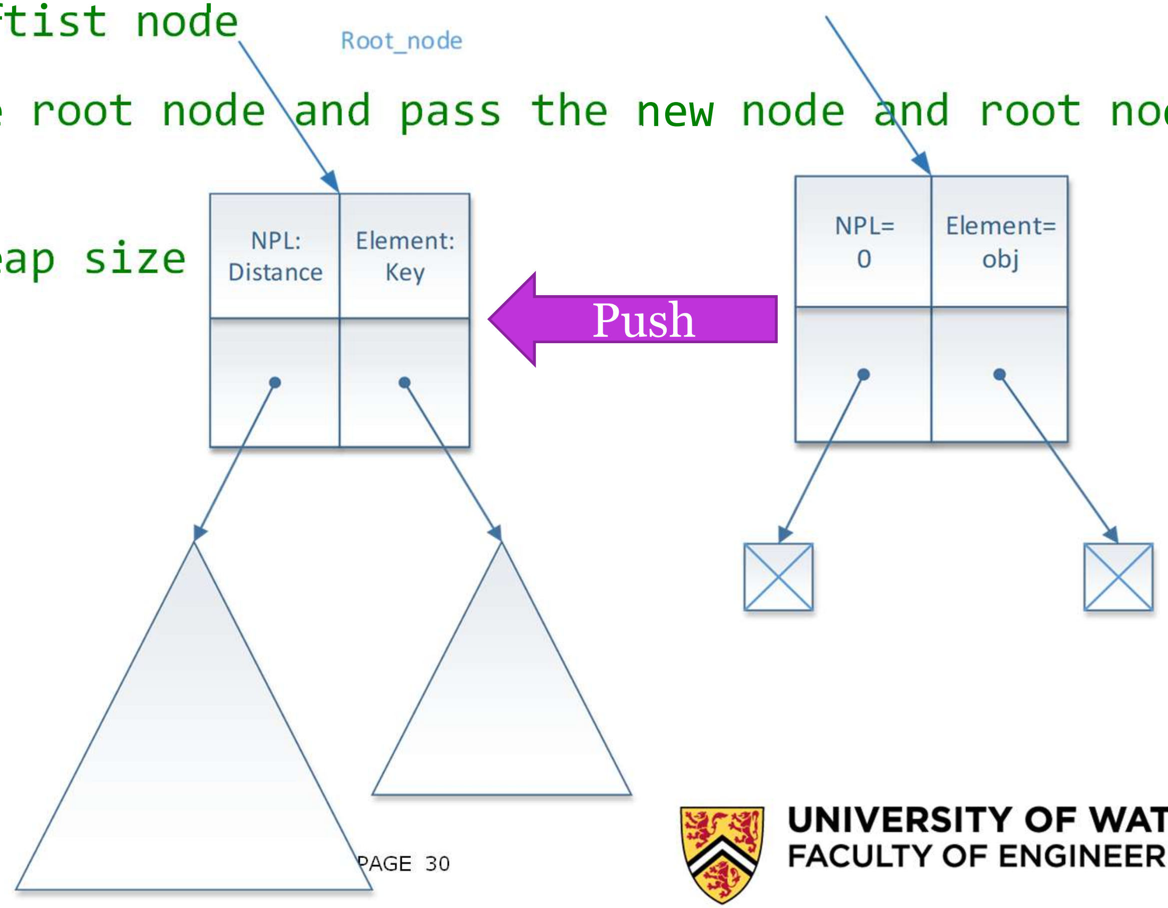
```
template<typename Type>
int Leftist_heap<Type>::count(Type const &obj) const {
// Return the number of instances of obj in the heap
}
```

Leftist_heap::clear()

```
template<typename Type>
void Leftist_heap<Type>::clear() {
// Call clear on the root node
// Reset the root node
// Reset the heap size
}
```

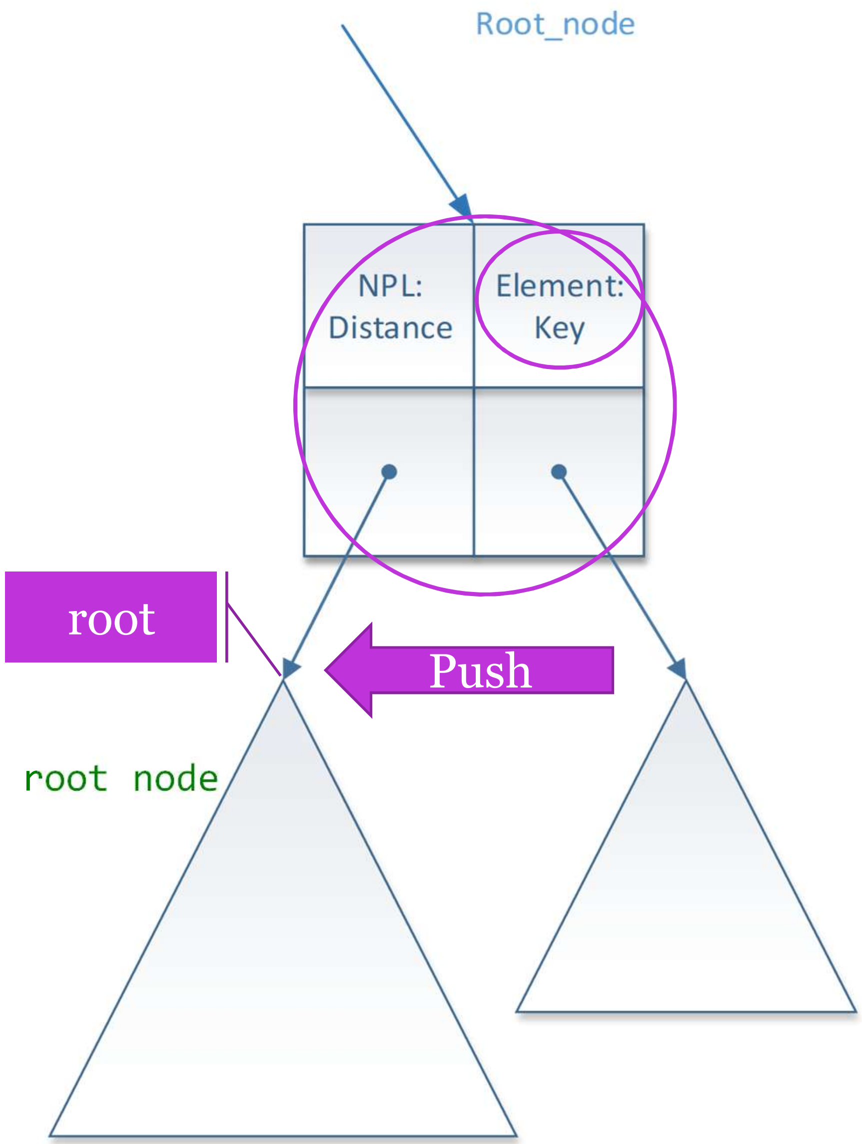
Leftist_heap::push()

```
template<typename Type>

void Leftist_heap<Type>::push(Type const &obj) {

// Create a new leftist node

// Call push on the root node and pass the new node and root node as
the arguments

//Increament the heap size

}
```



Leftist_heap::pop()

```
template<typename Type>
Type Leftist_heap<Type>::pop() {
// If the heap is empty throw underflow
// Pop the last element and delete its node
// The left sub-tree becomes the root node
// The right sub-tree is pushed into the new root node
// Decrement the heap size
// Return the element of the popped node

}
```



Leftist Heap

- Visualization:
  - https://www.cs.usfca.edu/~galles/visualization/LeftistHeap.html
- References
  - https://ece.uwaterloo.ca/~dwharder/aads/Algorithms/Leftist_heaps/
  - http://www.cs.cmu.edu/~clo/www/CMU/DataStructures/
  - http://www.dgp.toronto.edu/people/JamesStewart/378notes/
  - https://en.wikipedia.org/wiki/Leftist_tree
  - Weiss, Mark A. *Data structures & algorithm analysis in C++*. Pearson Education, 2012.