MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

10 Elementary Data Structures

Department of Computing and Software

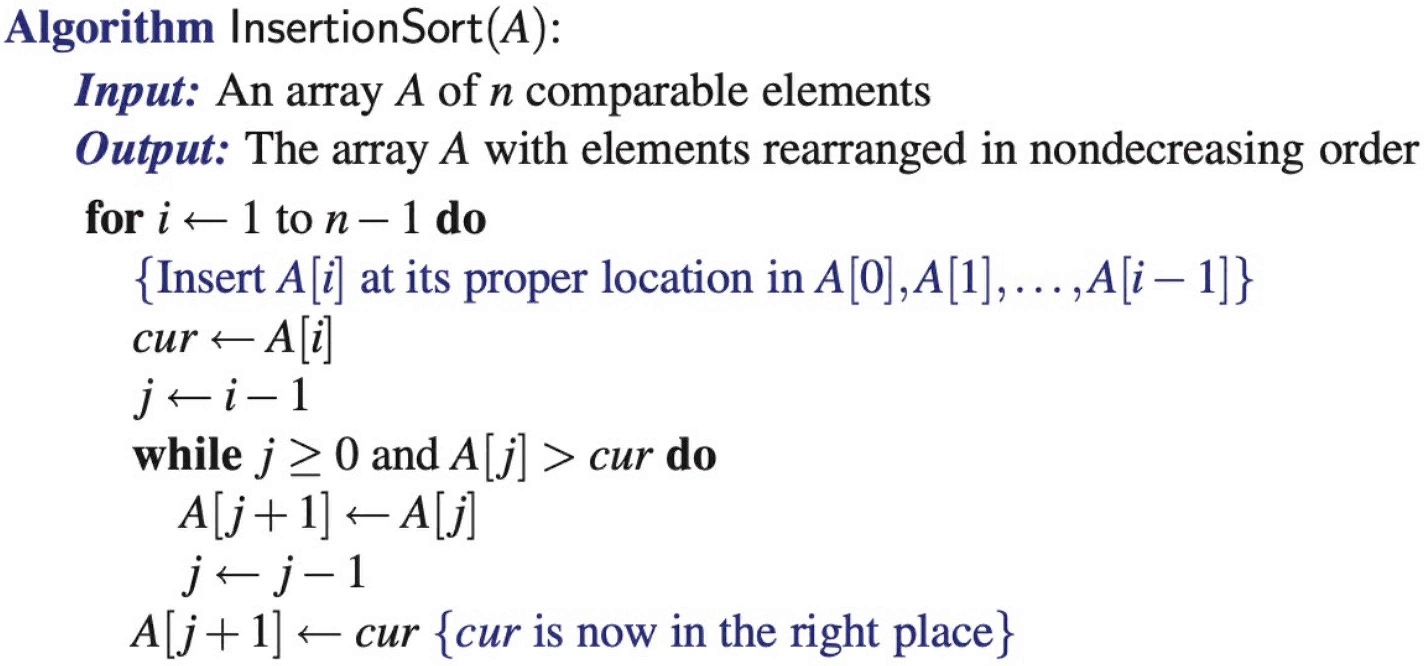
Instructor:

Omid Isfahanialamdari

February 3, 2022

Insertion Sort

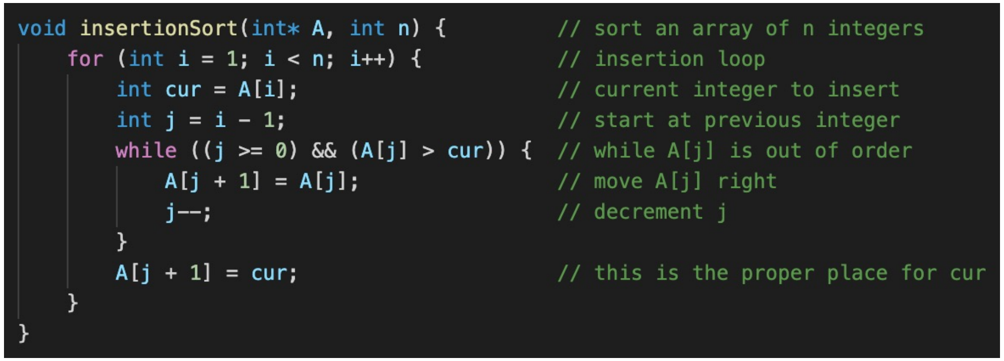
- The outer **for** loop considers each element in the array in turn
- The inner **while** loop moves that element to its proper location
- Always considers the subarray of elements that are to its left are sorted
- How to swap is important



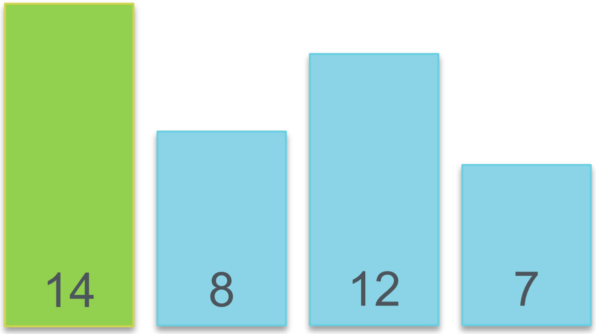
```
Algorithm InsertionSort($A$):
    Input: An array $A$ of $n$ comparable elements
    Output: The array $A$ with elements rearranged in nondecreasing order
    for $i \leftarrow 1$ to $n-1$ do
        {Insert $A[i]$ at its proper location in $A[0], A[1], \dots, A[i-1]$}
        $cur \leftarrow A[i]$
        $j \leftarrow i-1$
        while $j \geq 0$ and $A[j] > cur$ do
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j-1$
        $A[j+1] \leftarrow cur$ {$cur$ is now in the right place}
```

Insertion Sort Example

- The first one (A[0]) is already sorted!

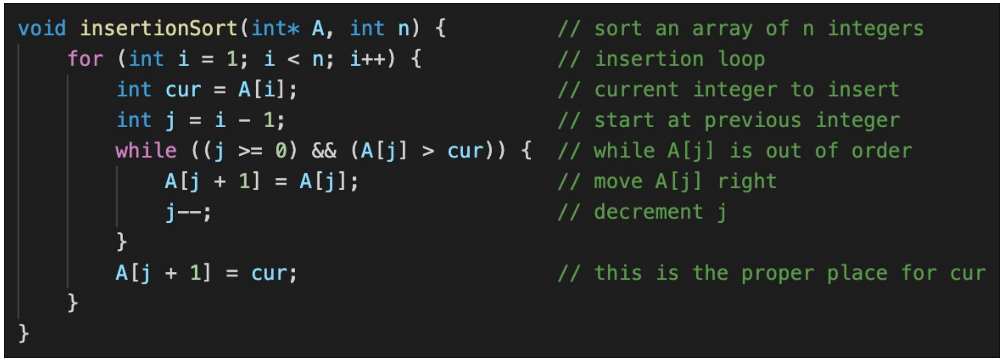


```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {  // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```

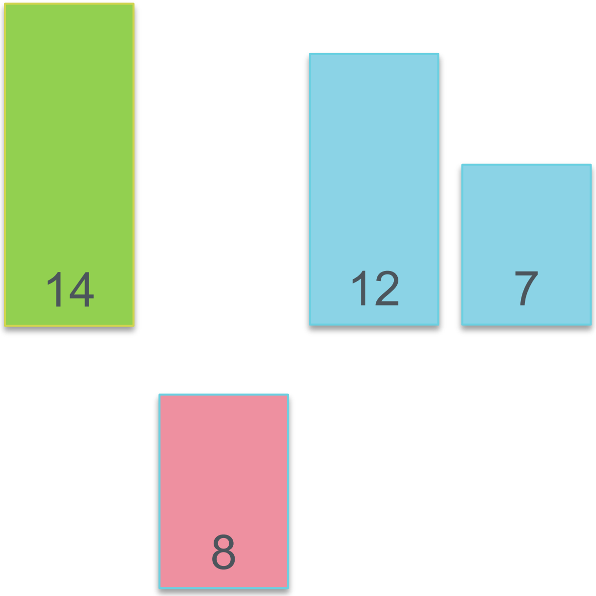


Insertion Sort Example

- i = 1
- cur = 8
- j = 0

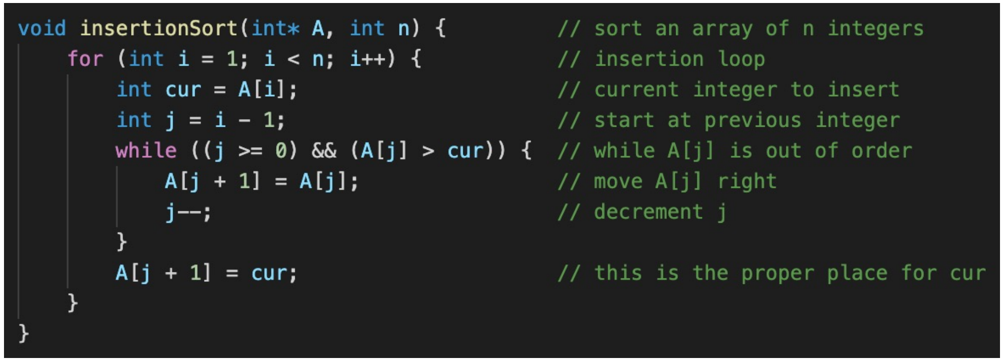


```
void insertionSort(int* A, int n) {         // sort an array of n integers
    for (int i = 1; i < n; i++) {           // insertion loop
        int cur = A[i];                     // current integer to insert
        int j = i - 1;                      // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {  // while A[j] is out of order
            A[j + 1] = A[j];                // move A[j] right
            j--;                            // decrement j
        }
        A[j + 1] = cur;                     // this is the proper place for cur
    }
}
```

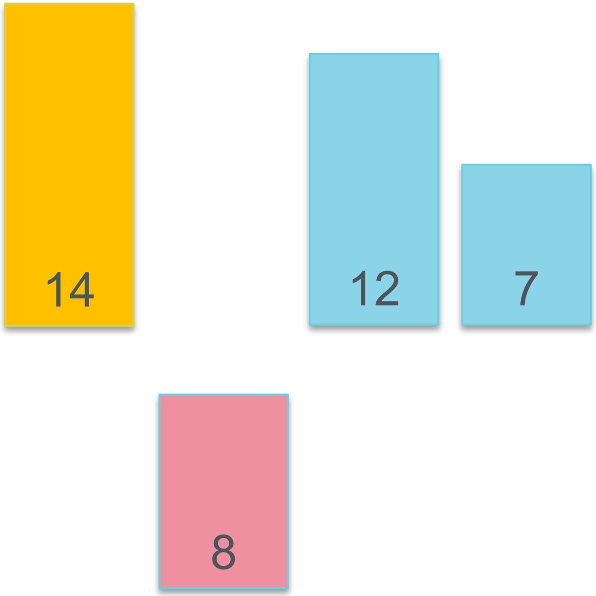


Insertion Sort Example

- cur = 8
- j = 0
- A[j] > cur

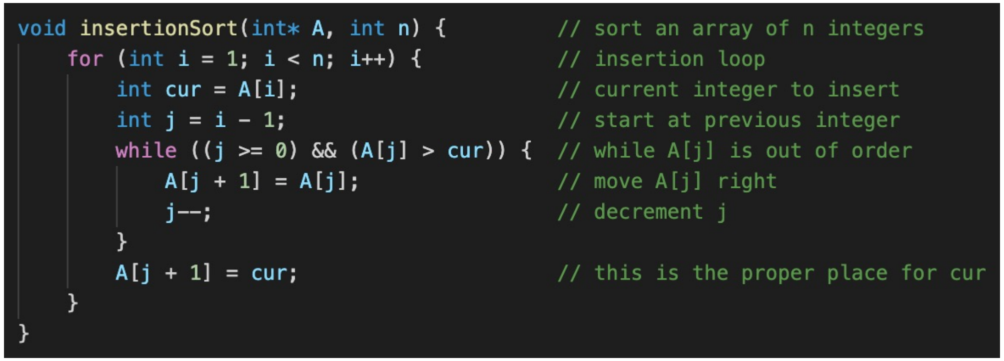


```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {   // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```

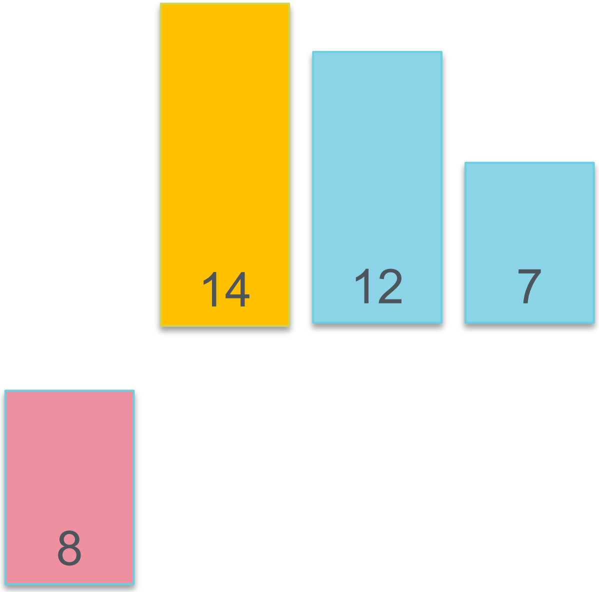


Insertion Sort Example

- A[j+1] = A[j]
- j --
- j is -1

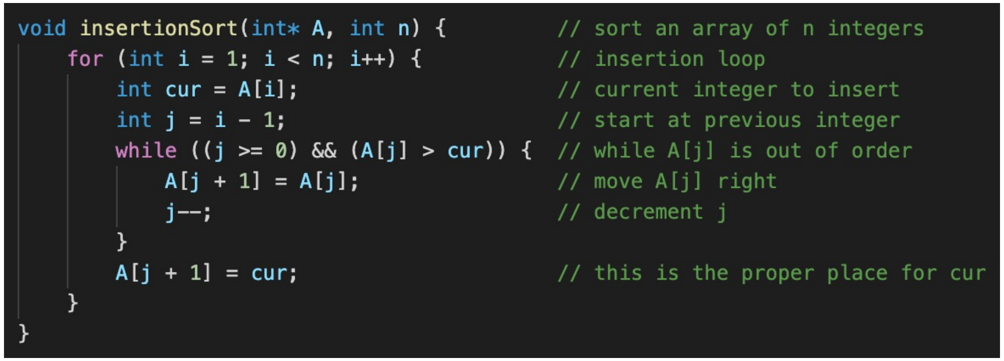


```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {   // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```

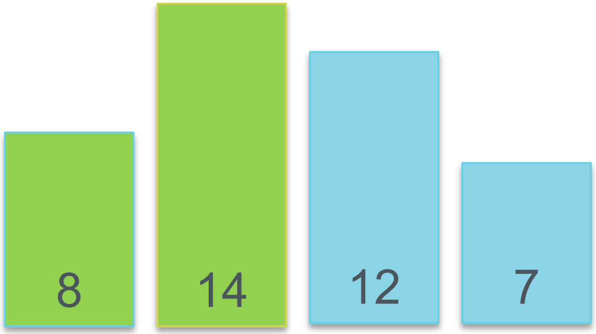


Insertion Sort Example

- A[j+1] = cur

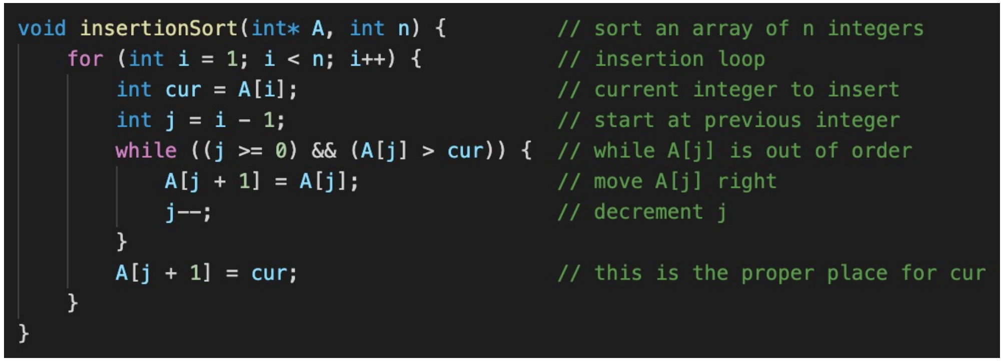


```
void insertionSort(int* A, int n) {           // sort an array of n integers
    for (int i = 1; i < n; i++) {             // insertion loop
        int cur = A[i];                       // current integer to insert
        int j = i - 1;                        // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {   // while A[j] is out of order
            A[j + 1] = A[j];                  // move A[j] right
            j--;                              // decrement j
        }
        A[j + 1] = cur;                       // this is the proper place for cur
    }
}
```

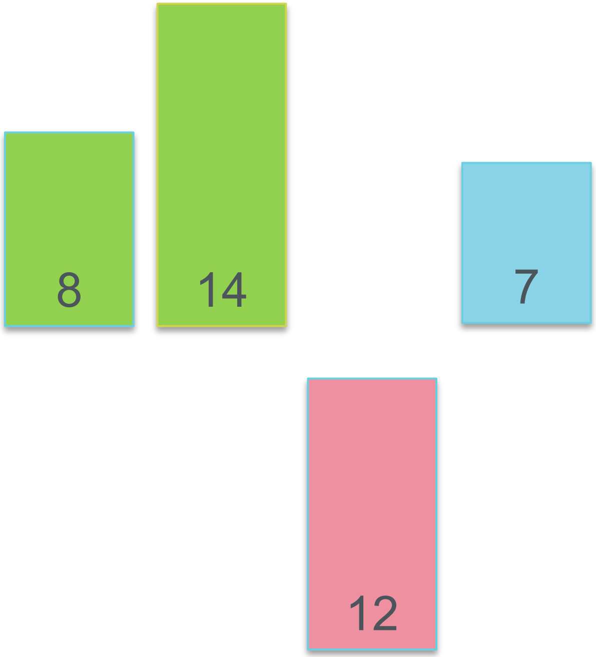


Insertion Sort Example

- i = 2
- cur = 12
- j = 1

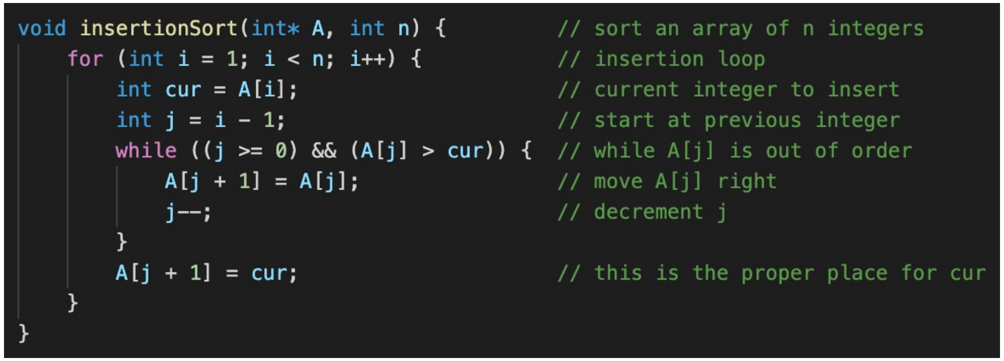


```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {   // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```

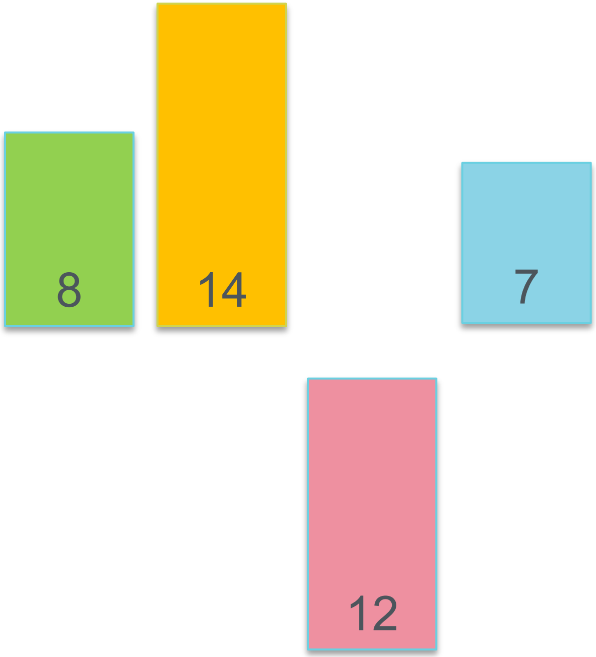


Insertion Sort Example

- cur = 12
- j = 1
- A[j] > cur

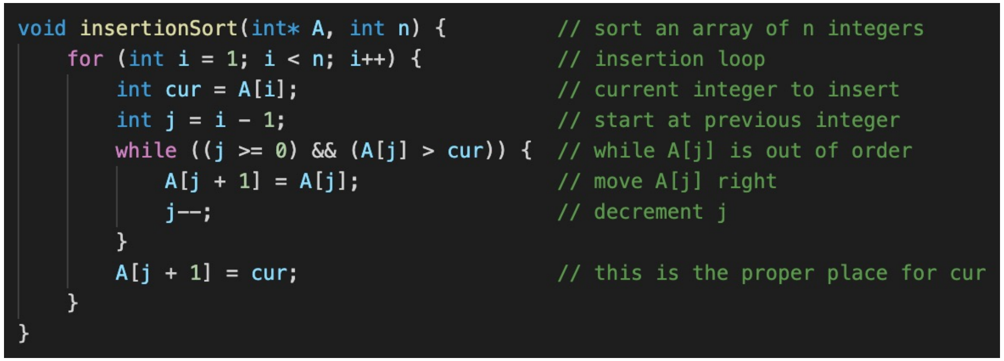


```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {  // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```

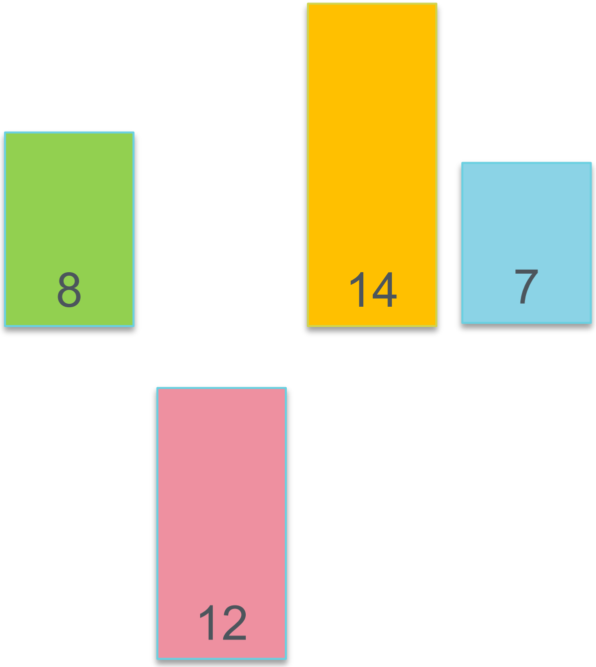


Insertion Sort Example

- A[j+1] = A[j]
- j --
- j is 0

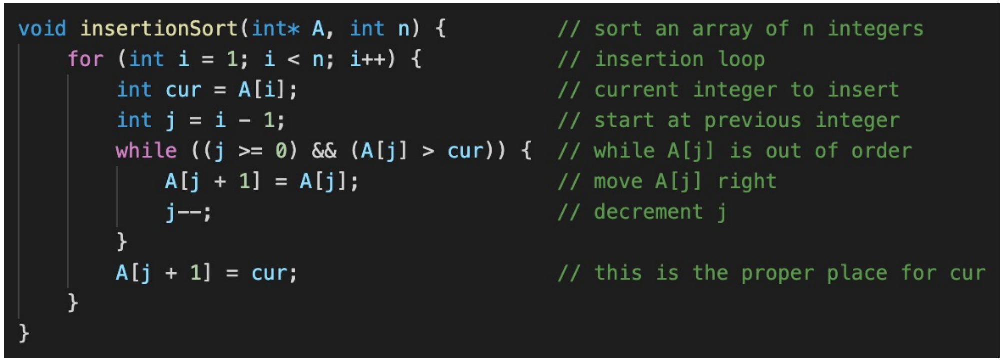


```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {   // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```

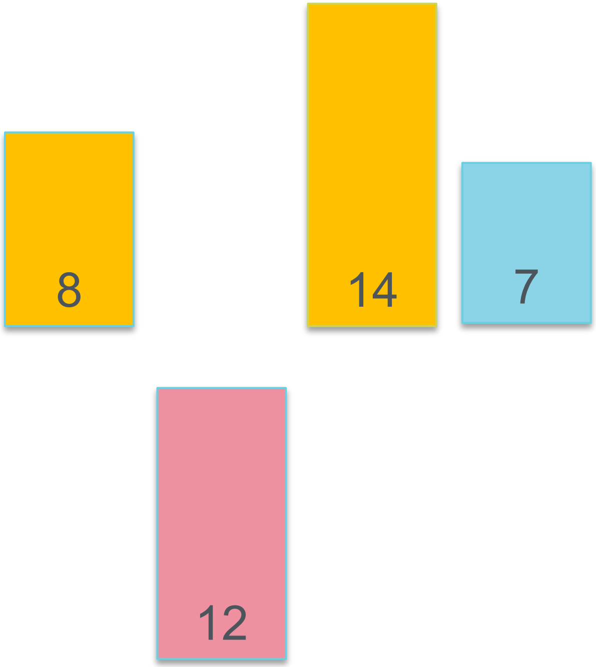


Insertion Sort Example

- cur = 12
- j = 0
- A[j] > cur? No!
  - while loop is skipped

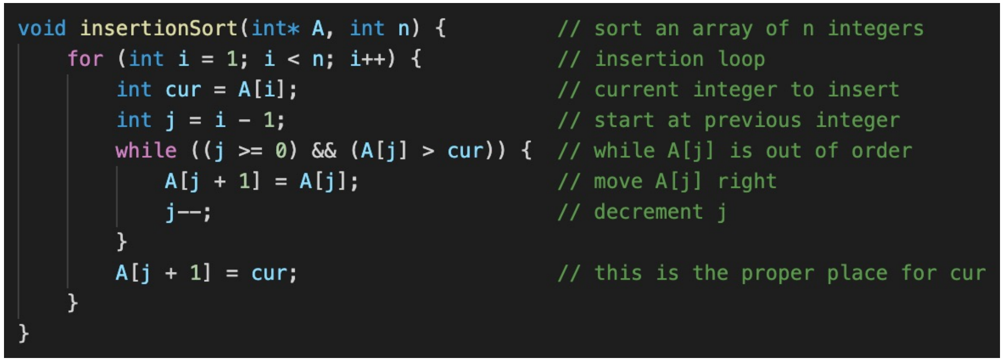


```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {   // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```

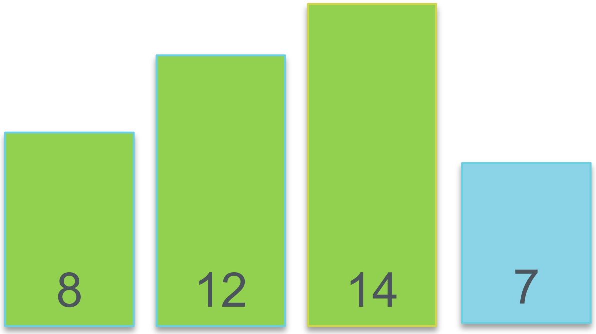


Insertion Sort Example

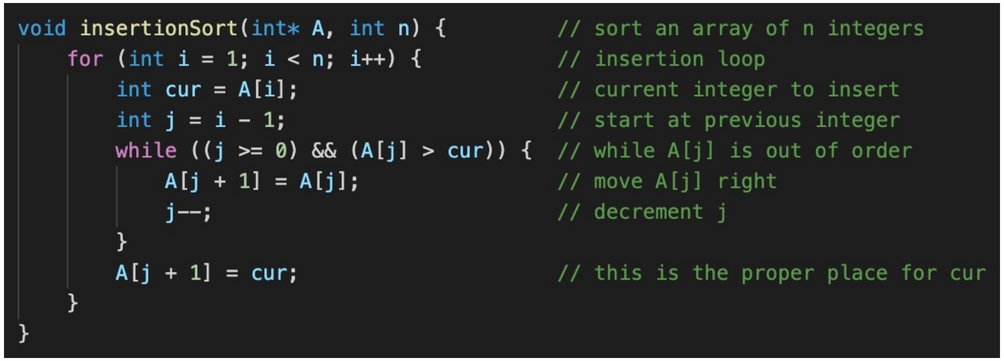
- j is 0
- cur is 12
- A[j+1] = cur
- Green items are already sorted



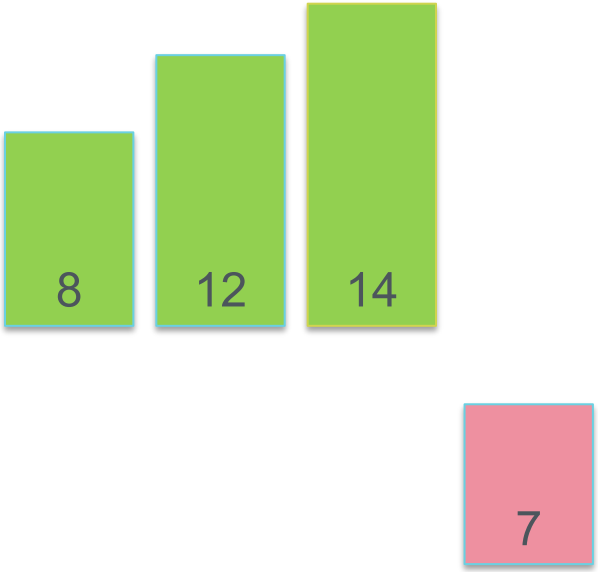
```
void insertionSort(int* A, int n) {         // sort an array of n integers
    for (int i = 1; i < n; i++) {           // insertion loop
        int cur = A[i];                     // current integer to insert
        int j = i - 1;                      // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {  // while A[j] is out of order
            A[j + 1] = A[j];                // move A[j] right
            j--;                            // decrement j
        }
        A[j + 1] = cur;                     // this is the proper place for cur
    }
}
```



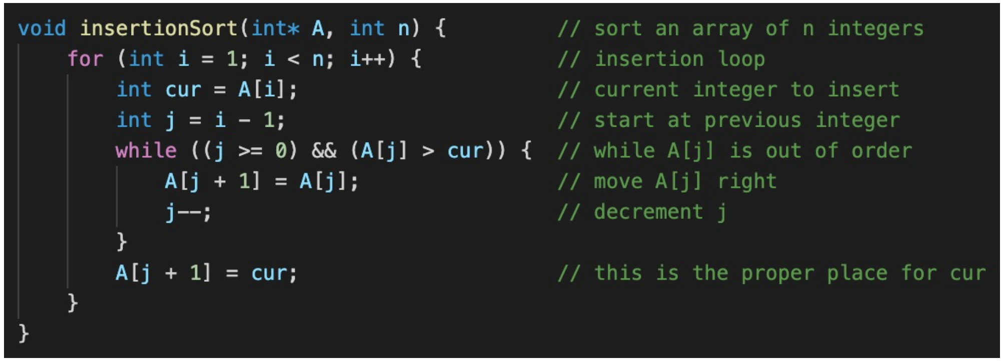
Insertion Sort Example



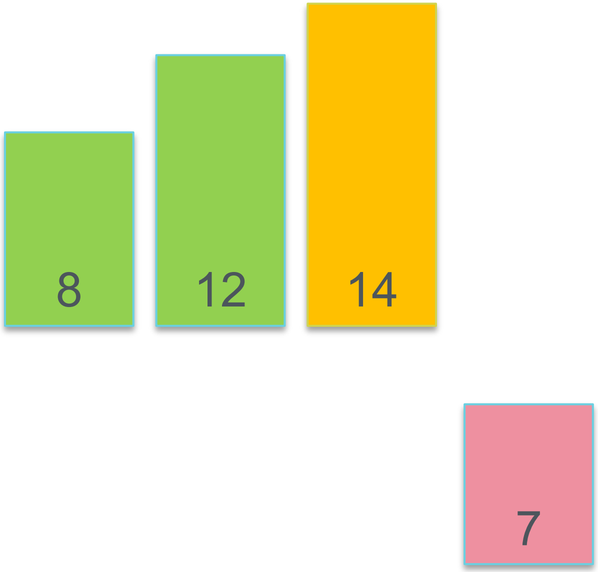
```
void insertionSort(int* A, int n) {           // sort an array of n integers
    for (int i = 1; i < n; i++) {              // insertion loop
        int cur = A[i];                        // current integer to insert
        int j = i - 1;                         // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {     // while A[j] is out of order
            A[j + 1] = A[j];                   // move A[j] right
            j--;                               // decrement j
        }
        A[j + 1] = cur;                        // this is the proper place for cur
    }
}
```



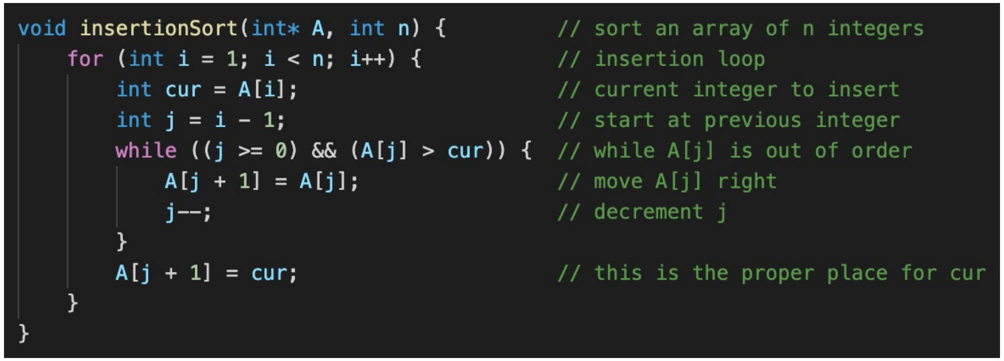
Insertion Sort Example



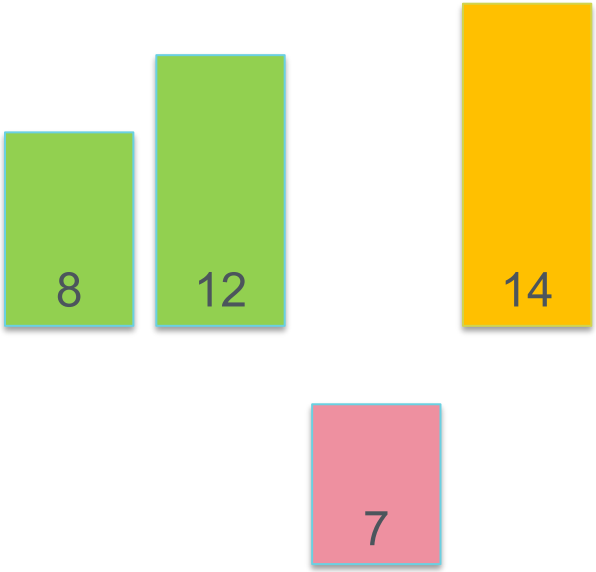
```
void insertionSort(int* A, int n) {         // sort an array of n integers
    for (int i = 1; i < n; i++) {           // insertion loop
        int cur = A[i];                     // current integer to insert
        int j = i - 1;                      // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {  // while A[j] is out of order
            A[j + 1] = A[j];                // move A[j] right
            j--;                            // decrement j
        }
        A[j + 1] = cur;                     // this is the proper place for cur
    }
}
```



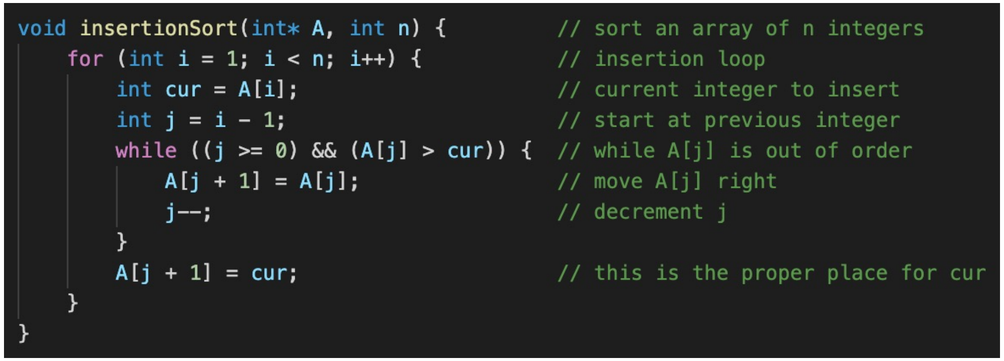
Insertion Sort Example



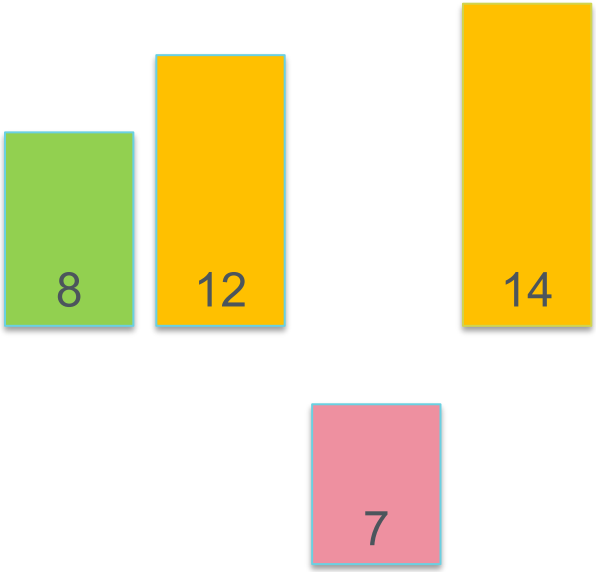
```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {  // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```



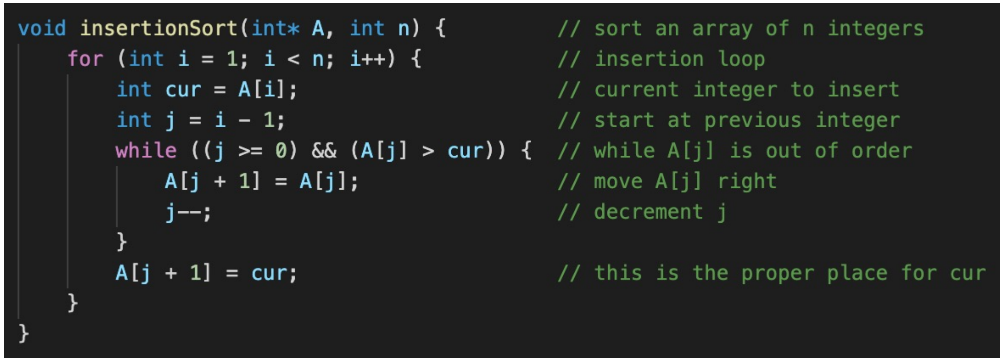
Insertion Sort Example



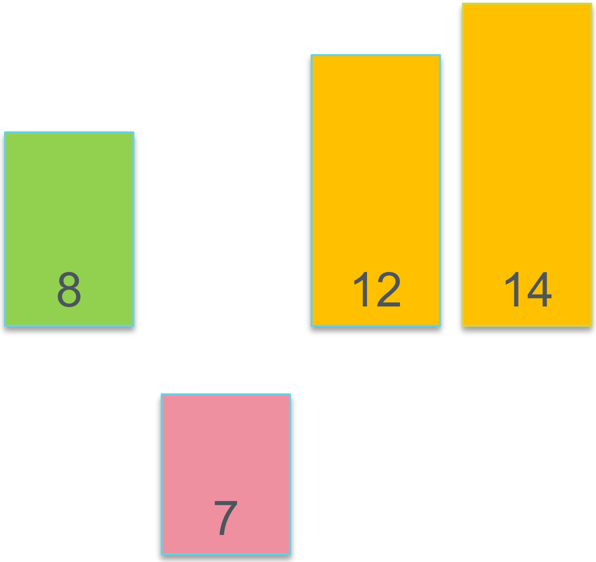
```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {   // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```



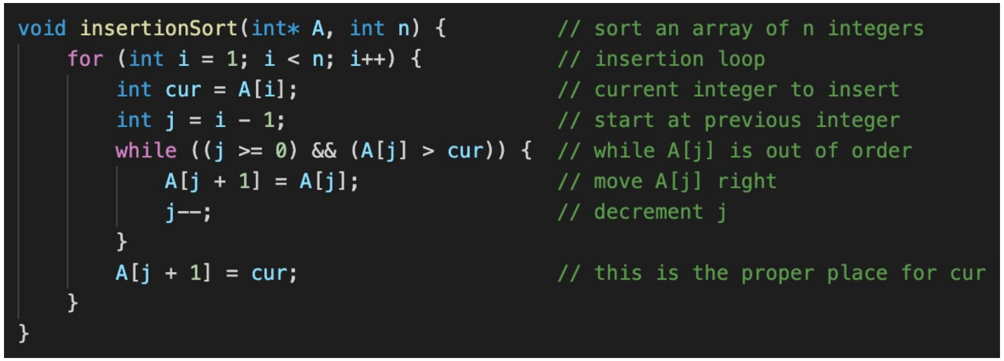
Insertion Sort Example



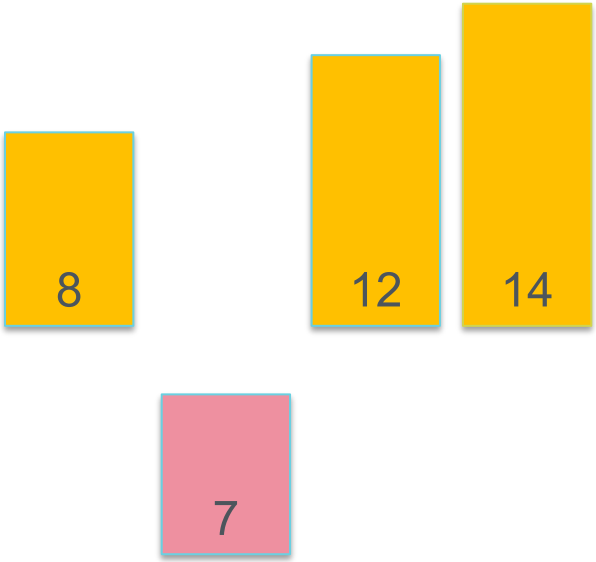
```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {  // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```



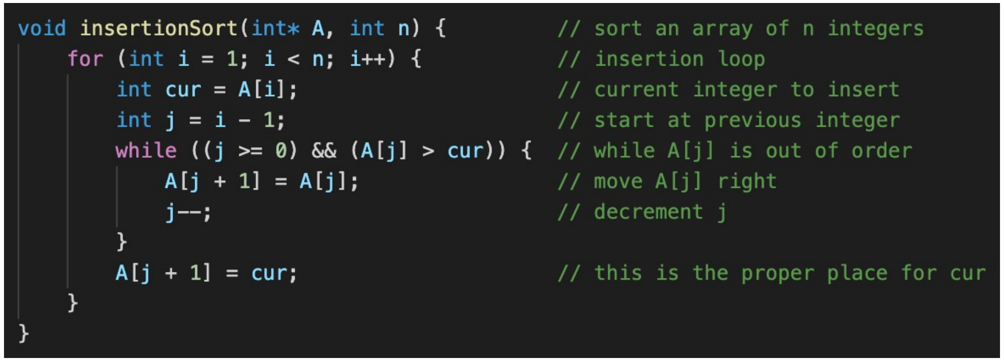
Insertion Sort Example



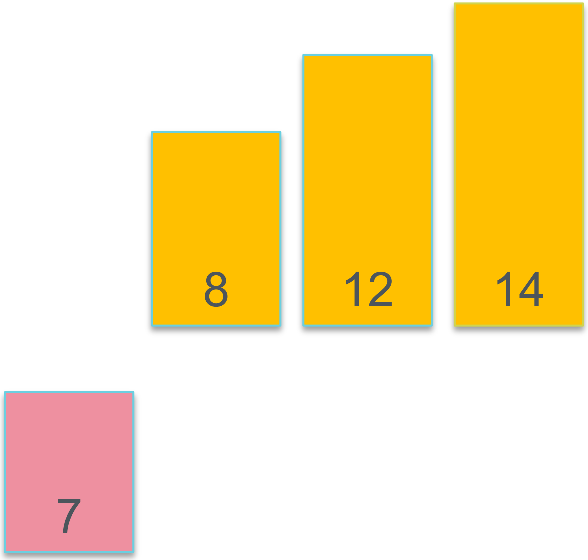
```
void insertionSort(int* A, int n) {         // sort an array of n integers
    for (int i = 1; i < n; i++) {           // insertion loop
        int cur = A[i];                     // current integer to insert
        int j = i - 1;                      // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {  // while A[j] is out of order
            A[j + 1] = A[j];                // move A[j] right
            j--;                            // decrement j
        }
        A[j + 1] = cur;                     // this is the proper place for cur
    }
}
```



Insertion Sort Example

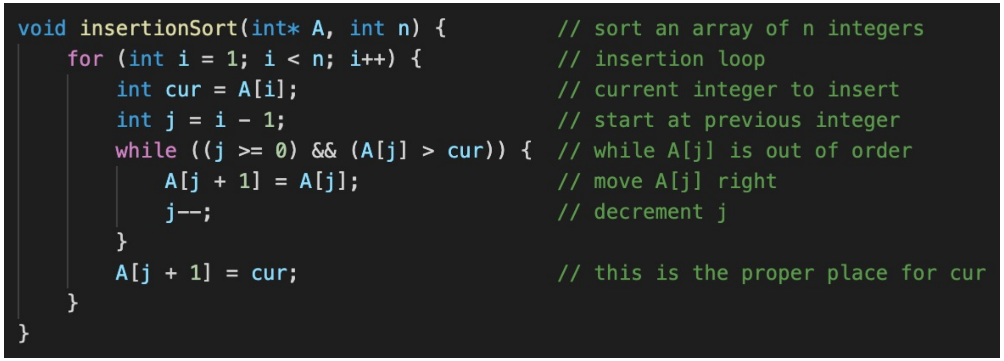


```
void insertionSort(int* A, int n) {         // sort an array of n integers
    for (int i = 1; i < n; i++) {           // insertion loop
        int cur = A[i];                     // current integer to insert
        int j = i - 1;                      // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {  // while A[j] is out of order
            A[j + 1] = A[j];                // move A[j] right
            j--;                            // decrement j
        }
        A[j + 1] = cur;                     // this is the proper place for cur
    }
}
```

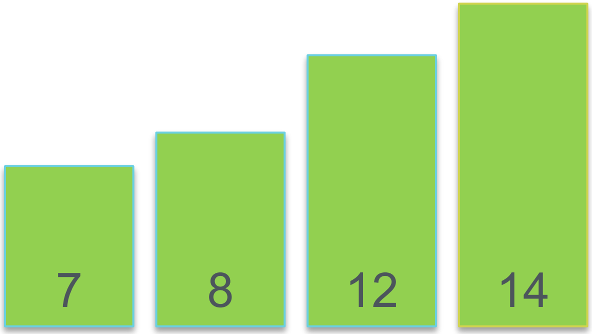


Insertion Sort Example

- Done!

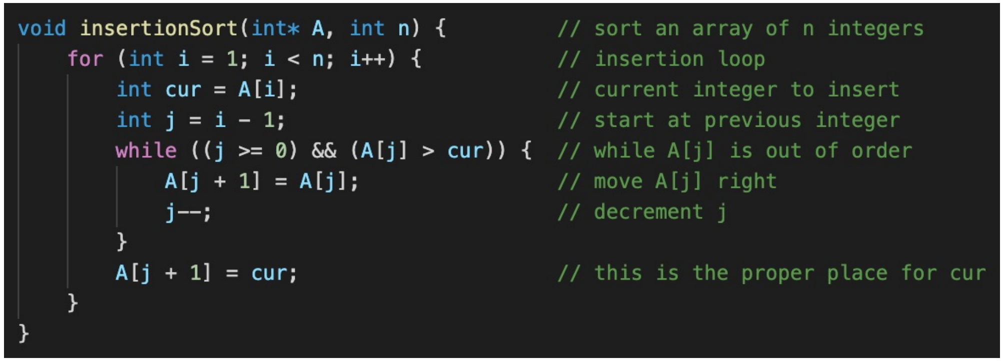


```
void insertionSort(int* A, int n) {            // sort an array of n integers
    for (int i = 1; i < n; i++) {              // insertion loop
        int cur = A[i];                        // current integer to insert
        int j = i - 1;                         // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {     // while A[j] is out of order
            A[j + 1] = A[j];                   // move A[j] right
            j--;                               // decrement j
        }
        A[j + 1] = cur;                        // this is the proper place for cur
    }
}
```

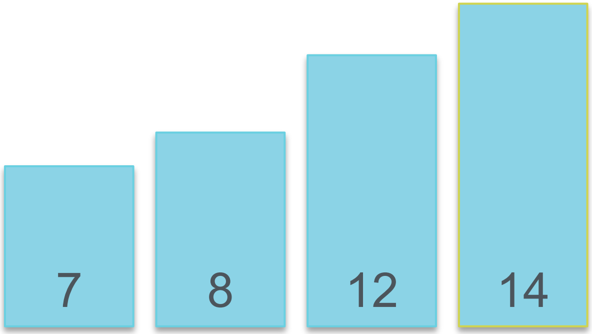


Insertion Sort Extreme Cases?

- What if the items are already sorted?

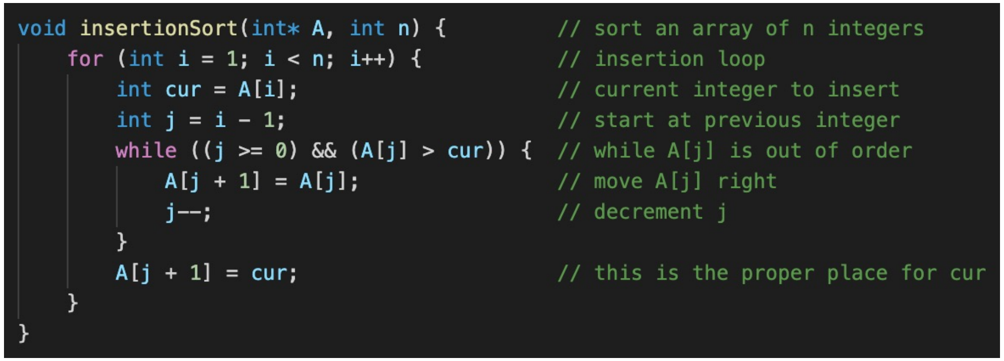


```
void insertionSort(int* A, int n) {           // sort an array of n integers
    for (int i = 1; i < n; i++) {             // insertion loop
        int cur = A[i];                       // current integer to insert
        int j = i - 1;                        // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {    // while A[j] is out of order
            A[j + 1] = A[j];                  // move A[j] right
            j--;                              // decrement j
        }
        A[j + 1] = cur;                       // this is the proper place for cur
    }
}
```

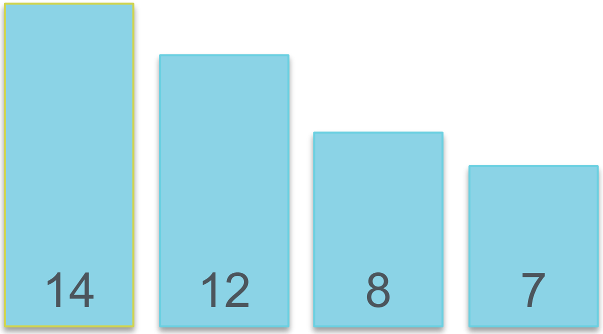


Insertion Sort Extreme Cases?

- What if the items are sorted decreasingly?
  - worst case!



```
void insertionSort(int* A, int n) {          // sort an array of n integers
    for (int i = 1; i < n; i++) {            // insertion loop
        int cur = A[i];                      // current integer to insert
        int j = i - 1;                       // start at previous integer
        while ((j >= 0) && (A[j] > cur)) {   // while A[j] is out of order
            A[j + 1] = A[j];                 // move A[j] right
            j--;                             // decrement j
        }
        A[j + 1] = cur;                      // this is the proper place for cur
    }
}
```



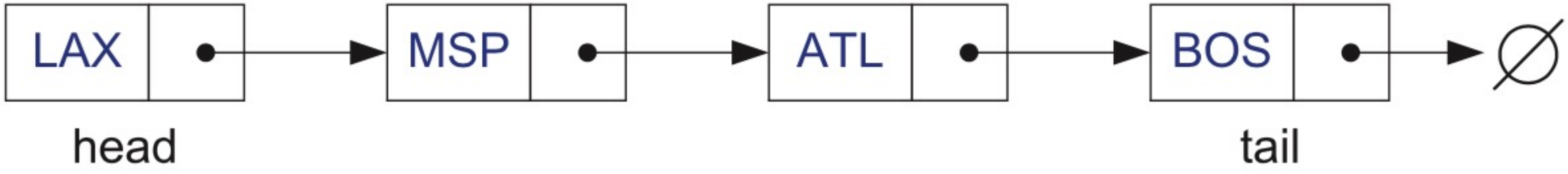
Analysis of Algorithms

- Later we will see how to analyze the behavior of algorithms under different conditions and reason about their complexity.
    - Worst case
    - Average case
    - Best case

Limitations of Arrays

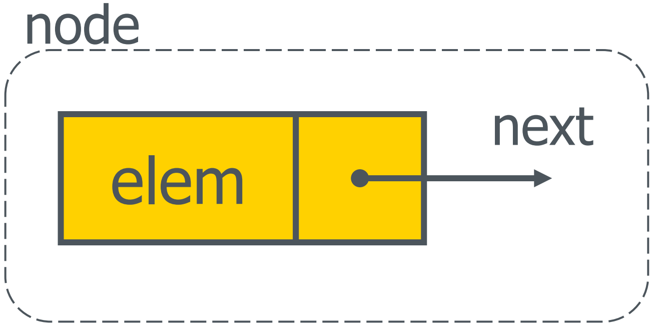
- Not adaptable, we must fix the size
  - Sometimes needed to be contiguous block of memory
- New insertion and deletion:
  - difficult Need to shift to make space for insertion
  - Need to fill empty positions after deletion

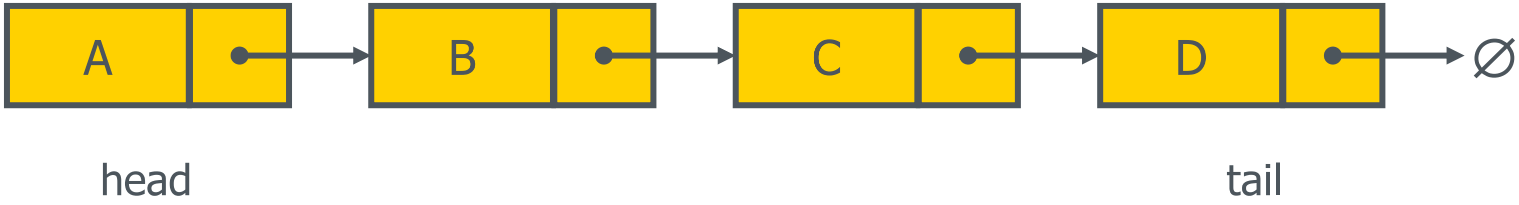
- Why don't we connect all elements just "logically" not "physically"?
  - Linked List



Singly Linked Lists

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node
- Order is determined by chain of *next* links
  - traverse by pointer hopping
- First node is called **head**
- Last node is called **tail** (has a **null** as next reference)
- No predefined fixed size!

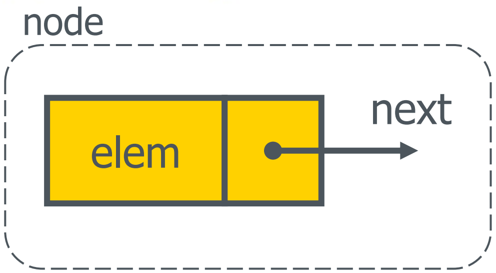




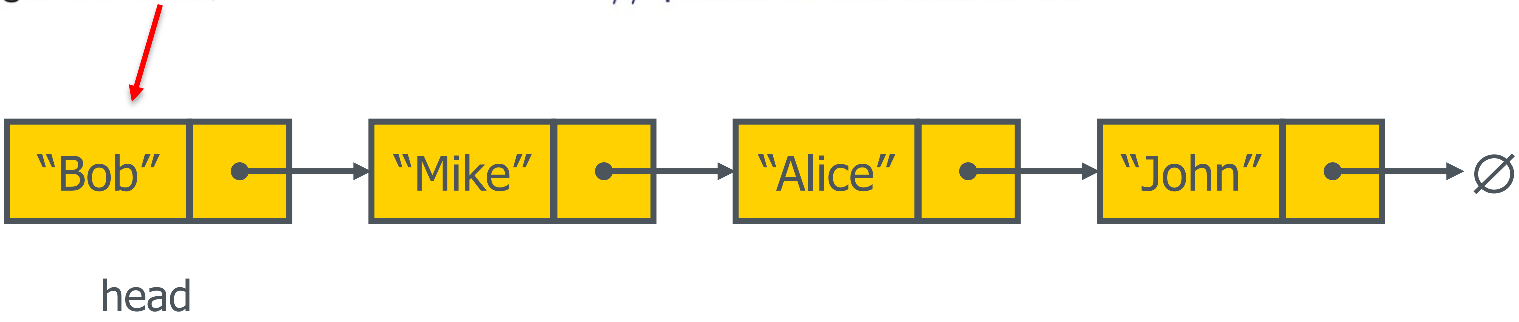
Singly Linked List C++ Classes Declaration

- For storing strings only!

```
class StringNode {                          // a node in a list of strings
private:
  string elem;                              // element value
  StringNode* next;                         // next item in the list

  friend class StringLinkedList;            // provide StringLinkedList access
};
```

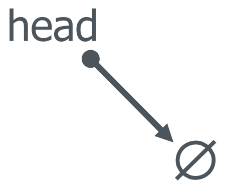


```
class StringLinkedList {                    // a linked list of strings
public:
  StringLinkedList();                       // empty list constructor
  ~StringLinkedList();                      // destructor
  bool empty() const;                       // is list empty?
  const string& front() const;              // get front element
  void addFront(const string& e);           // add to front of list
  void removeFront();                       // remove front item list
private:
  StringNode* head;                         // pointer to the head of list
};
```

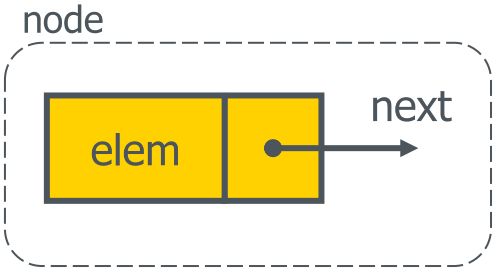


Singly Linked List Definitions

- Constructor
  - Set head to Null



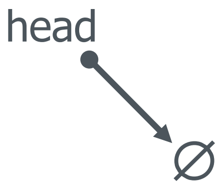
```
StringLinkedList::StringLinkedList()                 // constructor
  : head(NULL) { }

StringLinkedList::~StringLinkedList()                // destructor
  { while (!empty()) removeFront(); }

bool StringLinkedList::empty() const                 // is list empty?
  { return head == NULL; }

const string& StringLinkedList::front() const        // get front element
  { return head->elem; }
```

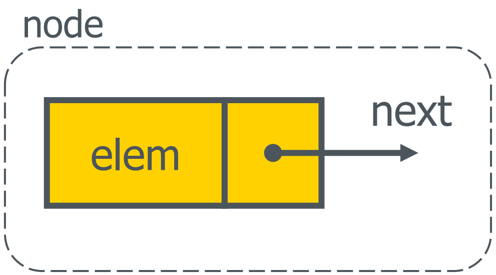


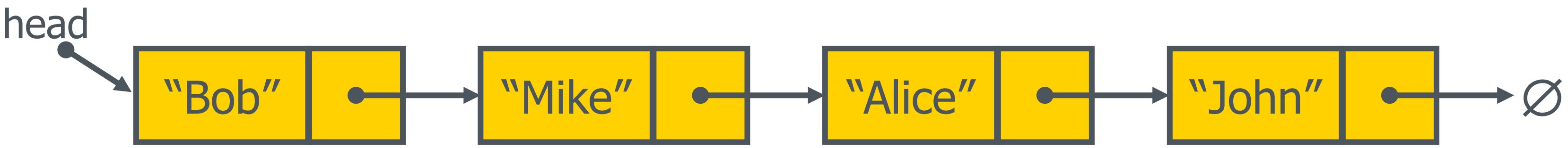
Singly Linked List Definitions

- Constructor
  - Set head to Null
- is Empty?
  - check if head is Null

```
StringLinkedList::StringLinkedList()                   // constructor
  : head(NULL) { }

StringLinkedList::~StringLinkedList()                  // destructor
  { while (!empty()) removeFront(); }

bool StringLinkedList::empty() const                   // is list empty?
  { return head == NULL; }

const string& StringLinkedList::front() const          // get front element
  { return head−>elem; }
```



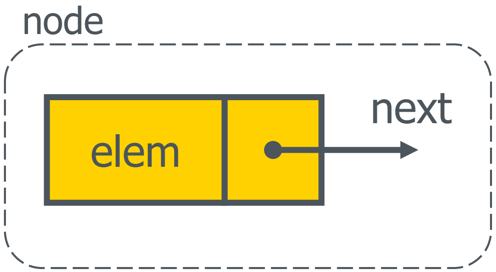


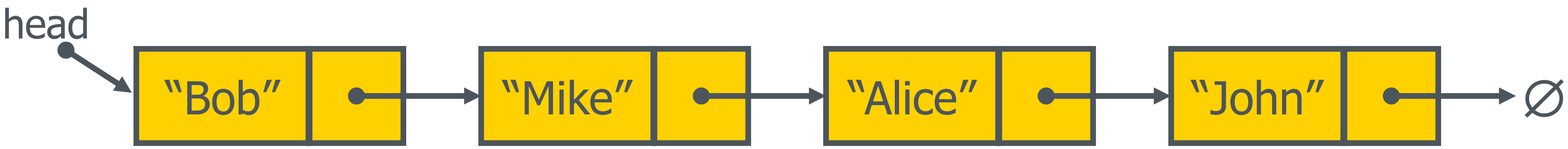


Singly Linked List Definitions

- Constructor
  - Set head to Null
- is Empty?
  - check if head is Null
- Return front element
  - return the element of the node head is pointing to

```
StringLinkedList::StringLinkedList()                  // constructor
  : head(NULL) { }

StringLinkedList::~StringLinkedList()                 // destructor
  { while (!empty()) removeFront(); }

bool StringLinkedList::empty() const                  // is list empty?
  { return head == NULL; }

const string& StringLinkedList::front() const         // get front element
  { return head->elem; }
```

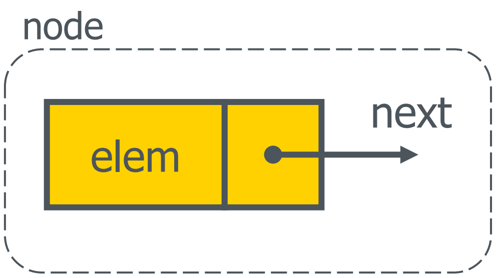


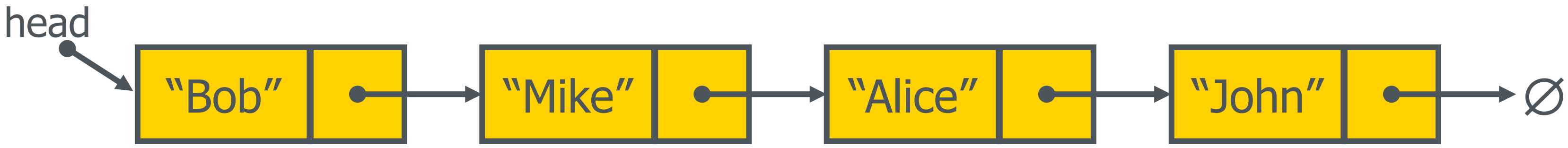


Singly Linked List Definitions

- Constructor
  - Set head to Null
- is Empty?
  - check if head is Null
- Return front element
  - return the element of the node head is pointing to
- Dynamic memory allocation
  - We need destructor
- Destructor
  - remove nodes until list is empty

```
StringLinkedList::StringLinkedList()                    // constructor
  : head(NULL) { }

StringLinkedList::~StringLinkedList()                   // destructor
  { while (!empty()) removeFront(); }

bool StringLinkedList::empty() const                    // is list empty?
  { return head == NULL; }

const string& StringLinkedList::front() const           // get front element
  { return head->elem; }
```

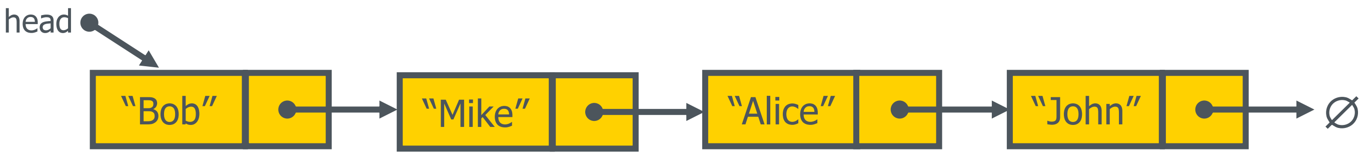




Singly Linked List - addFront

- Insert element at the head of the singly linked list

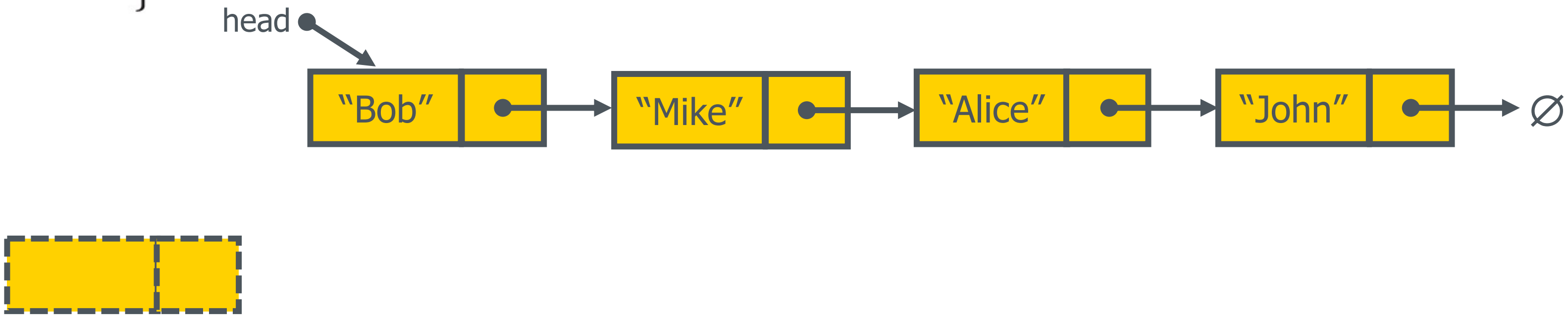
```
void StringLinkedList::addFront(const string& e) {   // add to front of list
  StringNode* v = new StringNode;                     // create new node
  v->elem = e;                                        // store data
  v->next = head;                                     // head now follows v
  head = v;                                           // v is now the head
}
```



Singly Linked List - addFront

- Insert element at the head of the singly linked list

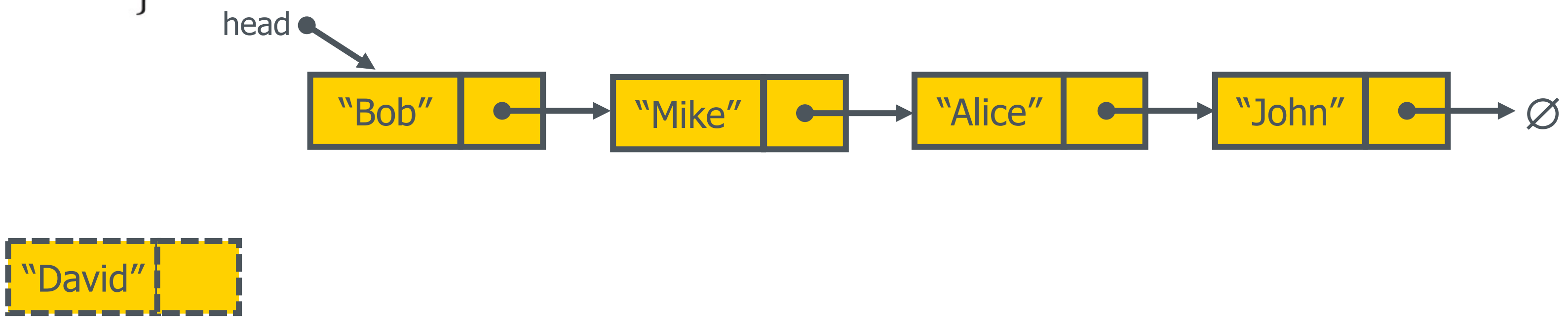
```
void StringLinkedList::addFront(const string& e) {  // add to front of list
  StringNode* v = new StringNode;                   // create new node
  v−>elem = e;                                      // store data
  v−>next = head;                                   // head now follows v
  head = v;                                         // v is now the head
}
```



Singly Linked List - addFront

- Insert element at the head of the singly linked list

```
void StringLinkedList::addFront(const string& e) {   // add to front of list
  StringNode* v = new StringNode;                    // create new node
  v−>elem = e;                                       // store data
  v−>next = head;                                    // head now follows v
  head = v;                                          // v is now the head
}
```

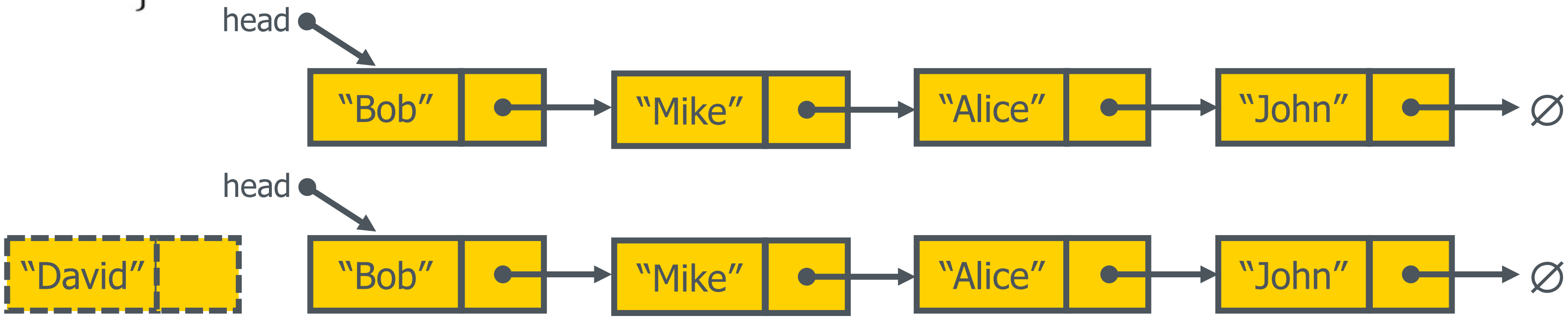


- Notice friendship!

Singly Linked List - addFront

- Insert element at the head of the singly linked list

```
void StringLinkedList::addFront(const string& e) {  // add to front of list
  StringNode* v = new StringNode;                    // create new node
  v->elem = e;                                       // store data
  v->next = head;                                    // head now follows v
  head = v;                                          // v is now the head
}
```

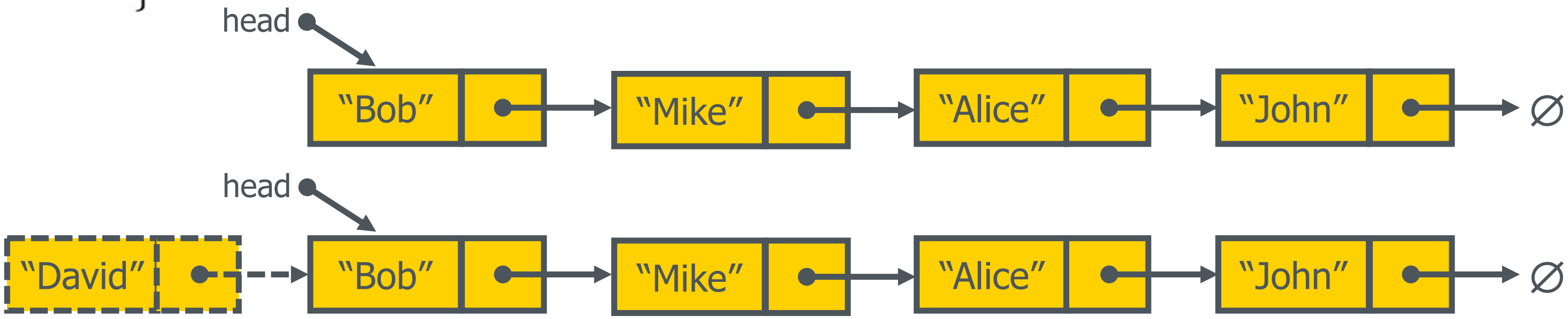


- Notice friendship!

Singly Linked List - addFront

- Insert element at the head of the singly linked list

```
void StringLinkedList::addFront(const string& e) {  // add to front of list
  StringNode* v = new StringNode;                    // create new node
  v->elem = e;                                       // store data
  v->next = head;                                    // head now follows v
  head = v;                                          // v is now the head
}
```

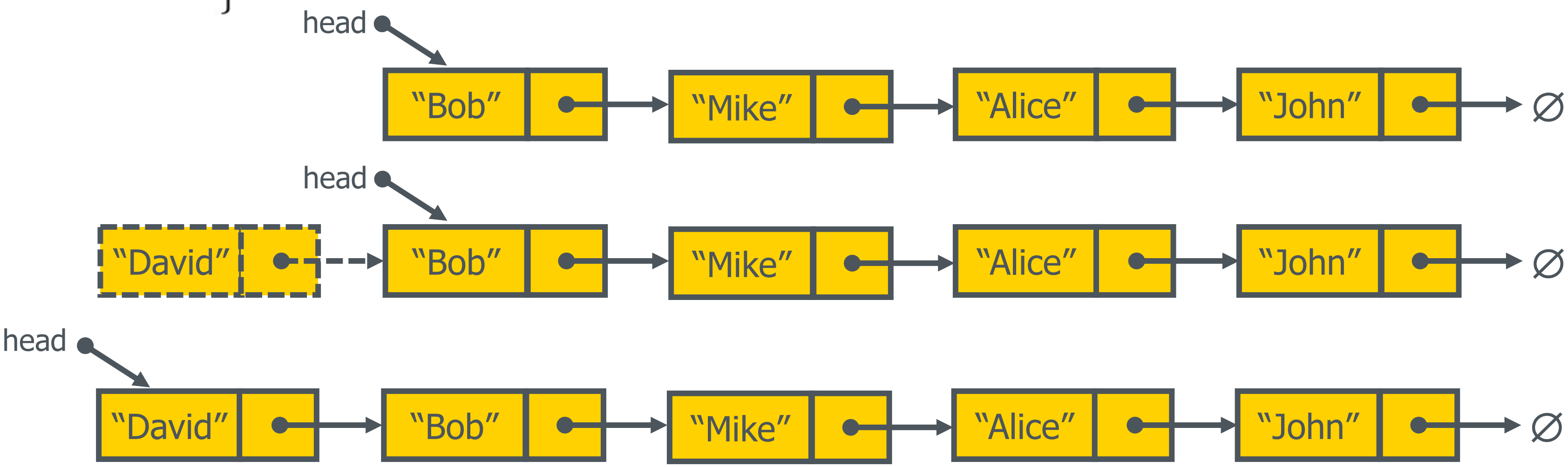


- Notice friendship!

Singly Linked List - addFront

- Insert element at the head of the singly linked list

```
void StringLinkedList::addFront(const string& e) {   // add to front of list
  StringNode* v = new StringNode;                     // create new node
  v->elem = e;                                        // store data
  v->next = head;                                     // head now follows v
  head = v;                                           // v is now the head
}
```

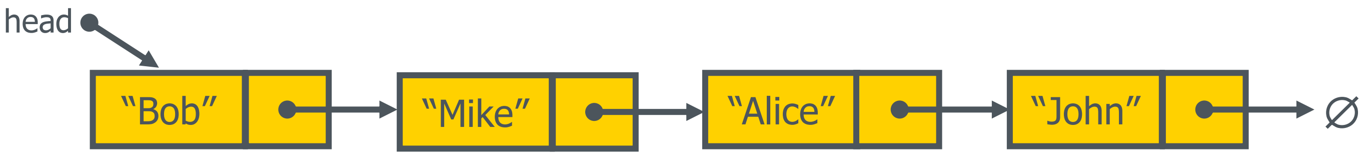


- Notice friendship!

Singly Linked List - removeFront

- Remove an element from the head of the singly linked list

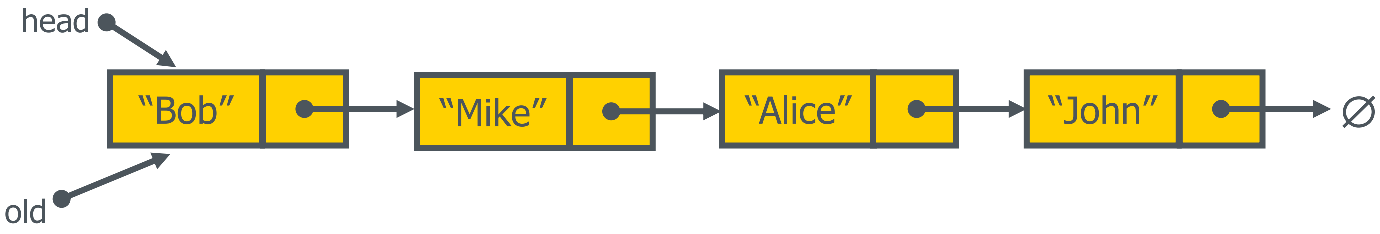
```
void StringLinkedList::removeFront() {          // remove front item
  StringNode* old = head;                       // save current head
  head = old−>next;                             // skip over old head
  delete old;                                   // delete the old head
}
```



Singly Linked List - removeFront

- Remove an element from the head of the singly linked list

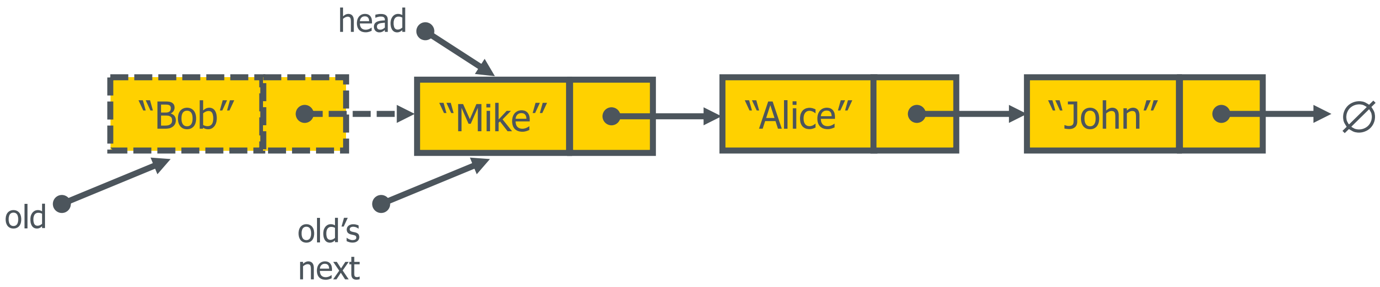
```
void StringLinkedList::removeFront() {          // remove front item
  StringNode* old = head;                       // save current head
  head = old->next;                             // skip over old head
  delete old;                                   // delete the old head
}
```



Singly Linked List - removeFront

- Remove an element from the head of the singly linked list

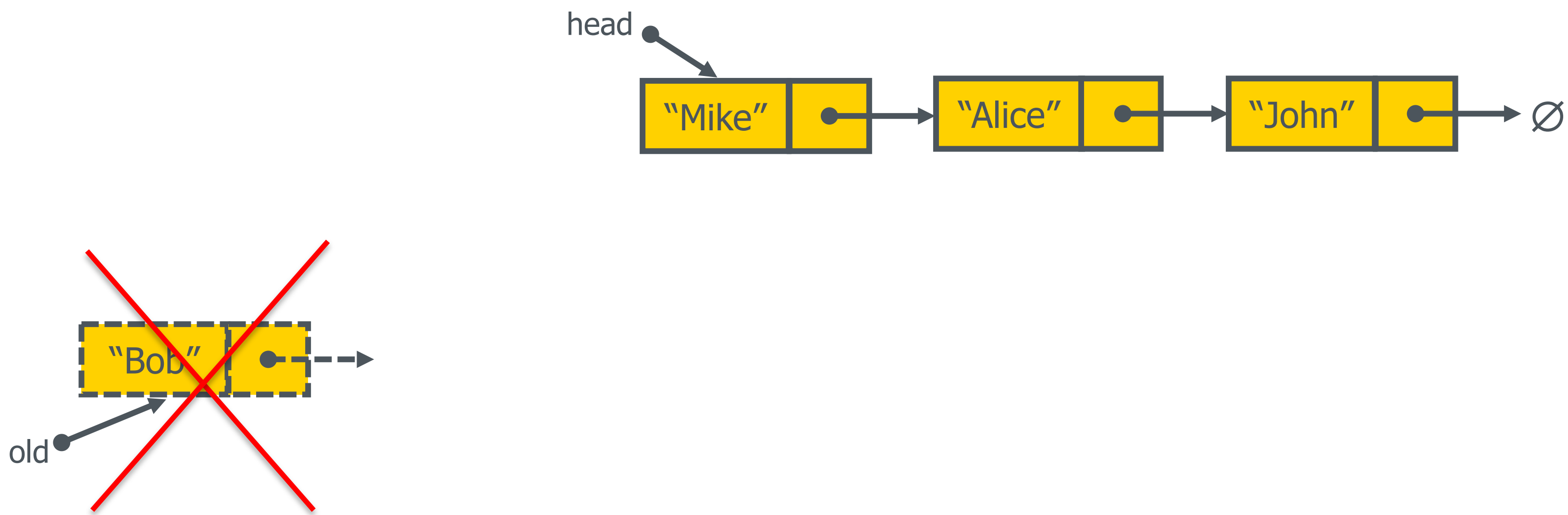
```
void StringLinkedList::removeFront() {          // remove front item
  StringNode* old = head;                       // save current head
  head = old->next;                             // skip over old head
  delete old;                                   // delete the old head
}
```



Singly Linked List - removeFront

- Remove an element from the head of the singly linked list

```
void StringLinkedList::removeFront() {          // remove front item
  StringNode* old = head;                        // save current head
  head = old−>next;                              // skip over old head
  delete old;                                    // delete the old head
}
```

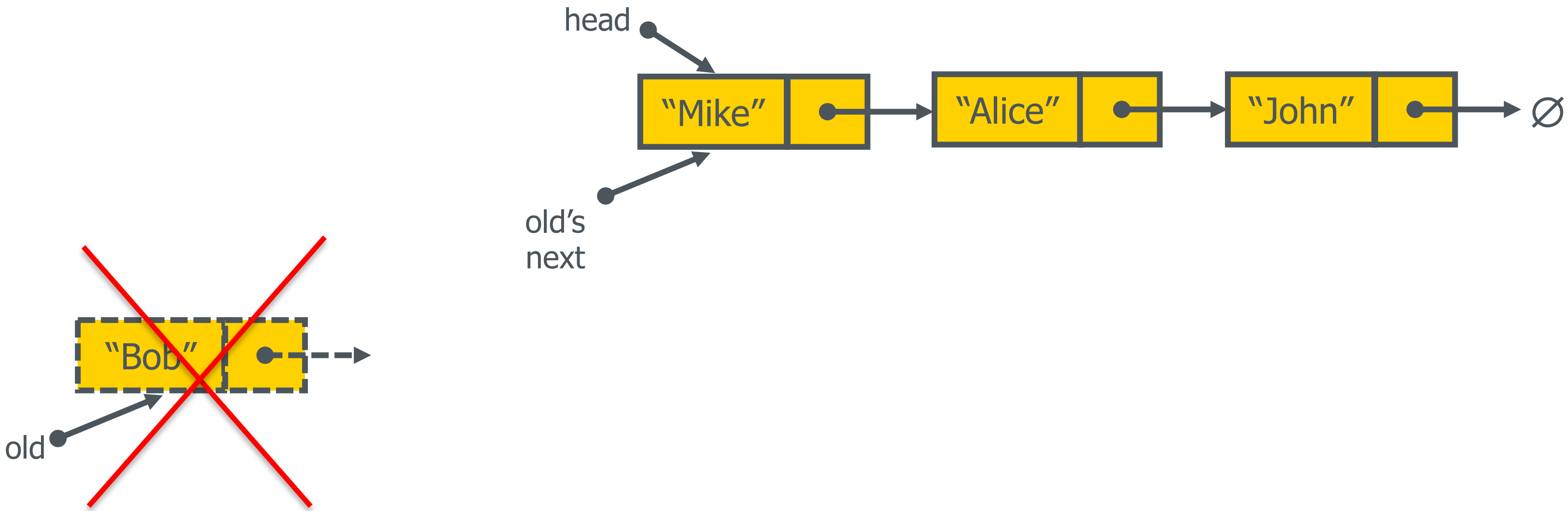


- Avoid memory leak!

Singly Linked List - removeFront

- Remove an element from the head of the singly linked list

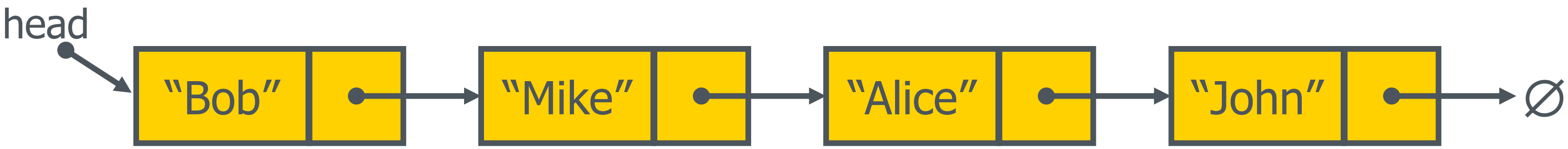
```
void StringLinkedList::removeFront() {      // remove front item
  StringNode* old = head;                   // save current head
  head = old−>next;                         // skip over old head
  delete old;                               // delete the old head
}
```



- Avoid memory leak!

Limitations of Singly Linked Lists

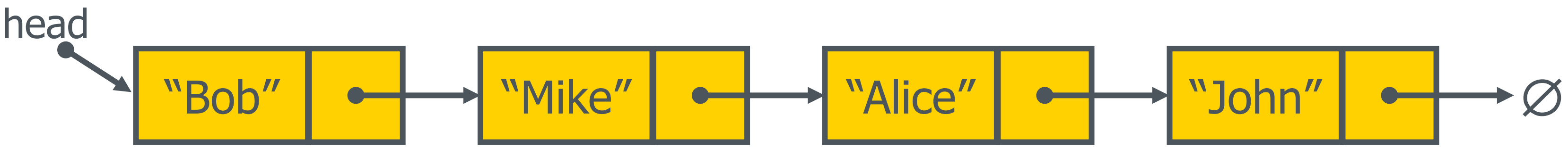
- Not easy to remove an element at the tail (or any other node)
  - ○ We don't have a quick way to access to the node immediately preceding the one we want to delete!



- How to insert at the tail?

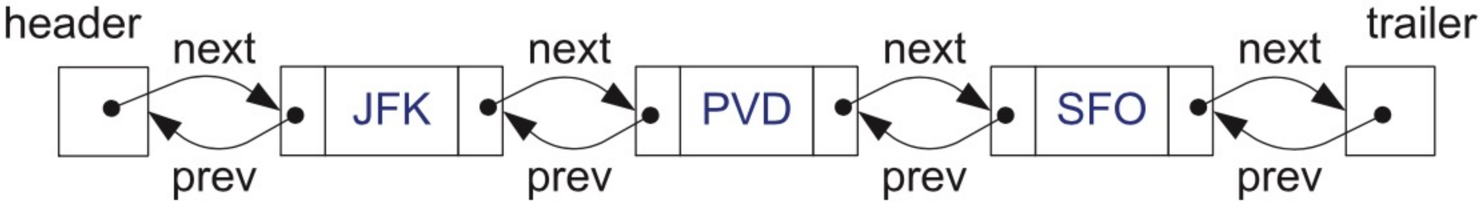
Limitations of Singly Linked Lists

- Not easy to remove an element at the tail (or any other node)
  - We don't have a quick way to access to the node immediately preceding the one we want to delete!



- How to insert at the tail?

- Better Data Structure
  - Doubly Linked List



Questions?