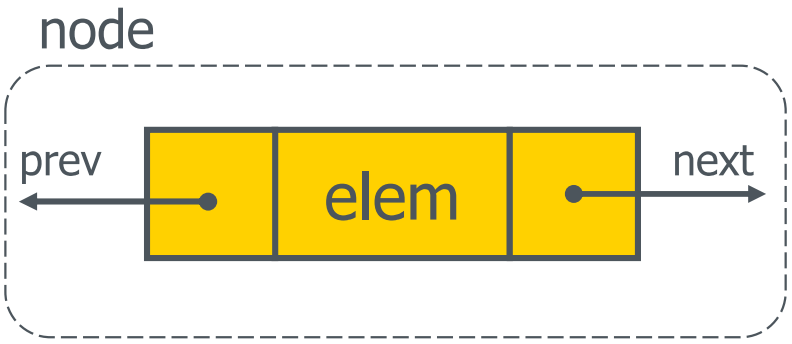
Doubly Linked Lists

- Allows traversing in both directions (forward and reverse)
- Each node stores
  - element
  - link to the next node
  - link to the previous node
- Sentinel nodes
  - Dummy **header** node
  - Dummy **trailer** node



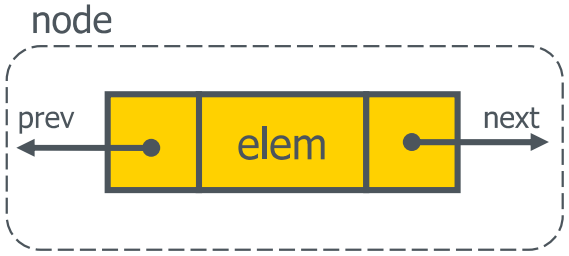
∅

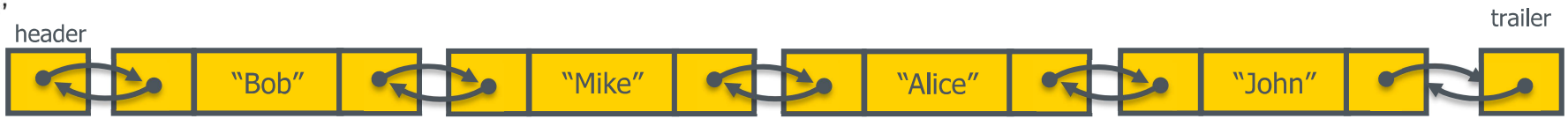
Doubly Linked List C++ Classes Declaration

- Notice typedef!
  - Generic, like templates

```
typedef string Elem;                    // list element type
class DNode {                           // doubly linked list node
private:
  Elem elem;                            // node element value
  DNode* prev;                          // previous node in list
  DNode* next;                          // next node in list
  friend class DLinkedList;             // allow DLinkedList access
};
```

```
class DLinkedList {                     // doubly linked list
public:
  DLinkedList();                        // constructor
  ~DLinkedList();                       // destructor
  bool empty() const;                   // is list empty?
  const Elem& front() const;            // get front element
  const Elem& back() const;             // get back element
  void addFront(const Elem& e);         // add to front of list
  void addBack(const Elem& e);          // add to back of list
  void removeFront();                   // remove from front
  void removeBack();                    // remove from back
private:                                // local type definitions
  DNode* header;                        // list sentinels
  DNode* trailer;
protected:                              // local utilities
  void add(DNode* v, const Elem& e);    // insert new node before v
  void remove(DNode* v);                // remove node v
};
```





Doubly Linked List Definitions

- Constructor
- is Empty?
  - header and trailer pointing each other
- Return front and back elements
- Dynamic memory allocation
  - We need destructor
- Destructor
  - remove nodes until list is empty

```
DLinkedList::DLinkedList() {                // constructor
  header = new DNode;                       // create sentinels
  trailer = new DNode;
  header->next = trailer;                   // have them point to each other
  trailer->prev = header;
}

bool DLinkedList::empty() const             // is list empty?
  { return (header->next == trailer); }

const Elem& DLinkedList::front() const      // get front element
  { return header->next->elem; }

const Elem& DLinkedList::back() const       // get back element
  { return trailer->prev->elem; }
```

```
DLinkedList::~DLinkedList() {               // destructor
  while (!empty()) removeFront();           // remove all but sentinels
  delete header;                            // remove the sentinels
  delete trailer;
}
```

Doubly Linked List - Add Element

- add() is protected
  - Utility function, why?

```
                                                    // insert new node before v
void DLinkedList::add(DNode* v, const Elem& e) {
  DNode* u = new DNode; u->elem = e; // create a new node for e
  u->next = v;                      // link u in between v
  u->prev = v->prev;                // ...and v->prev
  v->prev->next = v->prev = u;
}

void DLinkedList::addFront(const Elem& e) // add to front of list
  { add(header->next, e); }

void DLinkedList::addBack(const Elem& e) // add to back of list
  { add(trailer, e); }
```

Doubly Linked List - Remove Element

- remove() is protected
  - Utility function

```
void DLinkedList::remove(DNode* v) {      // remove node v
  DNode* u = v->prev;                     // predecessor
  DNode* w = v->next;                     // successor
  u->next = w;                            // unlink v from list
  w->prev = u;
  delete v;
}

void DLinkedList::removeFront()           // remove from font
  { remove(header->next); }

void DLinkedList::removeBack()            // remove from back
  { remove(trailer->prev); }
```

Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

12 Linked Lists - Continued

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

February 9, 2022

Administration

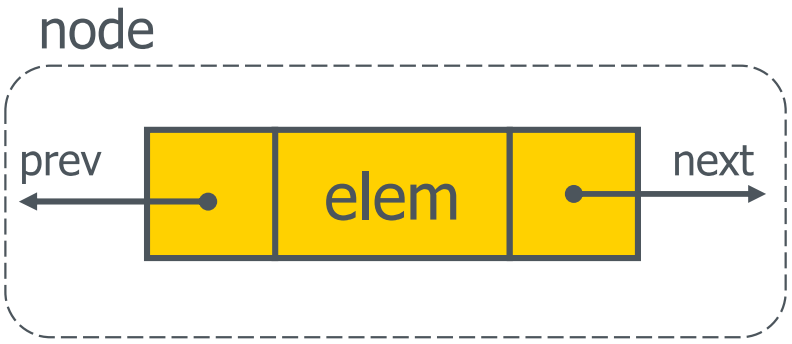
- The midterm will be in-person
  - Location: PGCLL B138
  - Date: Monday, February 14, 2022
  - Time: 1:30 PM - 2:30 PM **Mid-Term 1's duration: ONE HOUR**

- The university policy is that we are back fully in person as of Feb 7th. If someone feels unsafe coming to campus to take a test, they can apply for an exemption for in person learning through student accessibility services, though these will only be approved if appropriate medical documentation is provided.

- Read the announcement about resources that you can study for the mid-term.
- My Office Hour:
  - Today at 15:00 in **ITB-159** in-person (or virtually using teams as usual)

Doubly Linked Lists

- Allows traversing in both directions (forward and reverse)
- Each node stores
  - element
  - link to the next node
  - link to the previous node
- Sentinel nodes
  - Dummy **header** node
  - Dummy **trailer** node



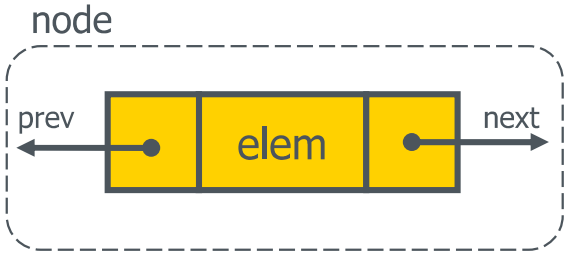
∅

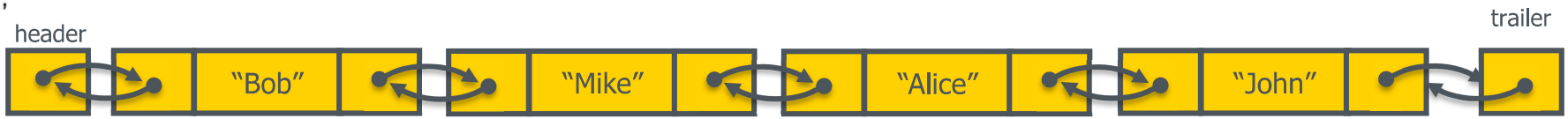
Doubly Linked List C++ Classes Declaration

- Notice typedef!
  - Generic, like templates

```
typedef string Elem;                    // list element type
class DNode {                           // doubly linked list node
private:
  Elem elem;                            // node element value
  DNode* prev;                          // previous node in list
  DNode* next;                          // next node in list
  friend class DLinkedList;             // allow DLinkedList access
};
```

```
class DLinkedList {                     // doubly linked list
public:
  DLinkedList();                        // constructor
  ~DLinkedList();                       // destructor
  bool empty() const;                   // is list empty?
  const Elem& front() const;            // get front element
  const Elem& back() const;             // get back element
  void addFront(const Elem& e);         // add to front of list
  void addBack(const Elem& e);          // add to back of list
  void removeFront();                   // remove from front
  void removeBack();                    // remove from back
private:                                // local type definitions
  DNode* header;                        // list sentinels
  DNode* trailer;
protected:                              // local utilities
  void add(DNode* v, const Elem& e);    // insert new node before v
  void remove(DNode* v);                // remove node v
};
```





Doubly Linked List Definitions

- Constructor
- is Empty?
  - header and trailer pointing each other
- Return front and back elements
- Dynamic memory allocation
  - We need destructor
- Destructor
  - remove nodes until list is empty

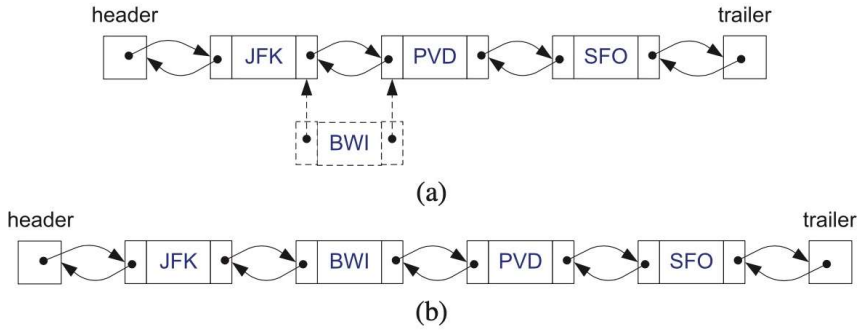
```
DLinkedList::DLinkedList() {                // constructor
  header = new DNode;                       // create sentinels
  trailer = new DNode;
  header->next = trailer;                   // have them point to each other
  trailer->prev = header;
}

bool DLinkedList::empty() const             // is list empty?
  { return (header->next == trailer); }

const Elem& DLinkedList::front() const      // get front element
  { return header->next->elem; }

const Elem& DLinkedList::back() const       // get back element
  { return trailer->prev->elem; }
```

```
DLinkedList::~DLinkedList() {               // destructor
  while (!empty()) removeFront();           // remove all but sentinels
  delete header;                            // remove the sentinels
  delete trailer;
}
```

Doubly Linked List - Add Element

- add() is protected
  - Utility function, why?

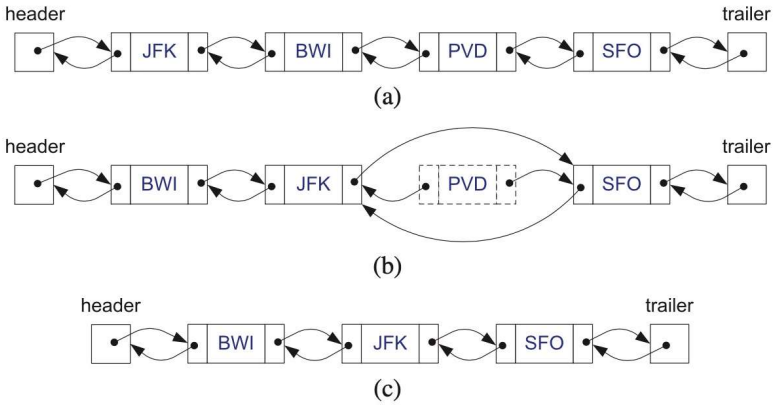
```
                                                    // insert new node before v
void DLinkedList::add(DNode* v, const Elem& e) {
  DNode* u = new DNode; u->elem = e;                // create a new node for e
  u->next = v;                                      // link u in between v
  u->prev = v->prev;                                // ...and v->prev
  v->prev->next = v->prev = u;
}

void DLinkedList::addFront(const Elem& e) // add to front of list
  { add(header->next, e); }

void DLinkedList::addBack(const Elem& e) // add to back of list
  { add(trailer, e); }
```



Doubly Linked List - Remove Element

- remove() is protected
  - Utility function

```
void DLinkedList::remove(DNode* v) {      // remove node v
  DNode* u = v->prev;                     // predecessor
  DNode* w = v->next;                     // successor
  u->next = w;                            // unlink v from list
  w->prev = u;
  delete v;
}

void DLinkedList::removeFront()           // remove from font
  { remove(header->next); }

void DLinkedList::removeBack()            // remove from back
  { remove(trailer->prev); }
```



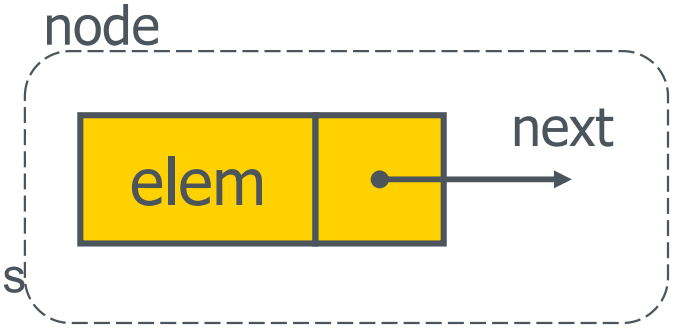
Doubly Linked List - Reverse

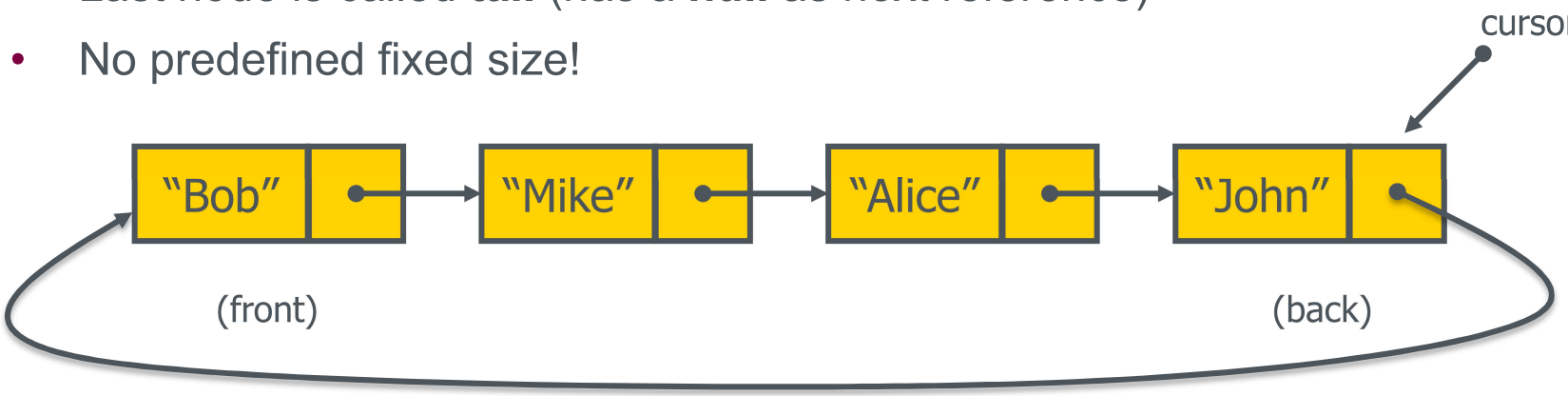
- This is not a function declared in the class
- See the tutorial of week 4 for a possible implementation in the class

```
void listReverse(DLinkedList& L) {          // reverse a list
  DLinkedList T;                            // temporary list
  while (!L.empty()) {                      // reverse L into T
    string s = L.front();  L.removeFront();
    T.addFront(s);
  }
  while (!T.empty()) {                      // copy T back to L
    string s = T.front();  T.removeFront();
    L.addBack(s);
  }
}
```

Circularly Linked List

- A variant of Singly Linked List
  - Rather than having a **head** or a **tail**, it forms a cycle
- Same node structure as Singly Linked List
- Cursor
  - A *virtual* starting node
  - This can be varying as we perform operations
- First node is called **head**
- Last node is called **tail** (has a **null** as next reference)
- No predefined fixed size!

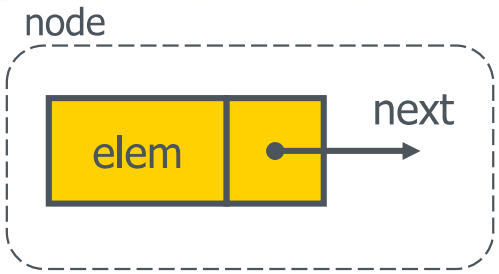




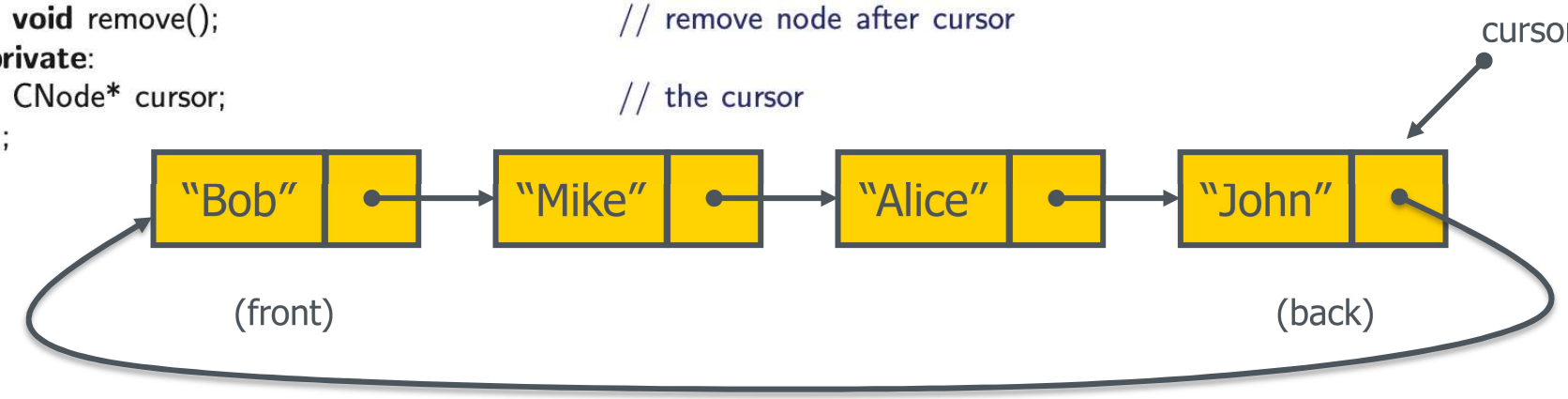
Circularly Linked List C++ Classes Declaration

- Notice typedef!
  - Generic, like templates

```
typedef string Elem;                  // element type
class CNode {                         // circularly linked list node
private:
  Elem elem;                          // linked list element value
  CNode* next;                        // next item in the list

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

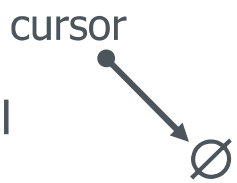



```
class CircleList {                    // a circularly linked list
public:
  CircleList();                       // constructor
  ~CircleList();                      // destructor
  bool empty() const;                 // is list empty?
  const Elem& front() const;          // element at cursor
  const Elem& back() const;           // element following cursor
  void advance();                     // advance cursor
  void add(const Elem& e);            // add after cursor
  void remove();                      // remove node after cursor
private:
  CNode* cursor;                      // the cursor
};
```

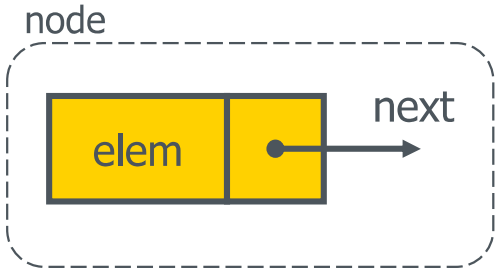


Circular Linked List Definitions

- Constructor
  - Set cursor to Null
- Destructor
  - remove nodes until list is empty



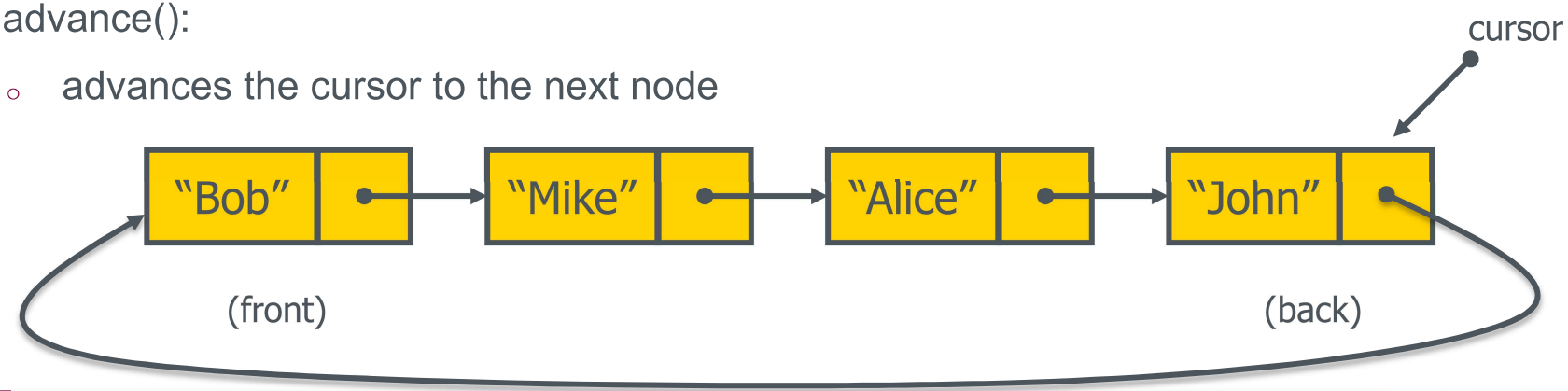

```
CircleList::CircleList()                    // constructor
  : cursor(NULL) { }
CircleList::~CircleList()                   // destructor
  { while (!empty()) remove(); }
```



Circular Linked List Definitions

- Constructor
  - Set cursor to Null
- Destructor
  - remove nodes until list is empty
- is Empty?
  - check if cursor is Null
- Return **front** element
  - return element immediately after cursor
- Return **back** element
  - return element referenced by cursor
- advance():
  - advances the cursor to the next node

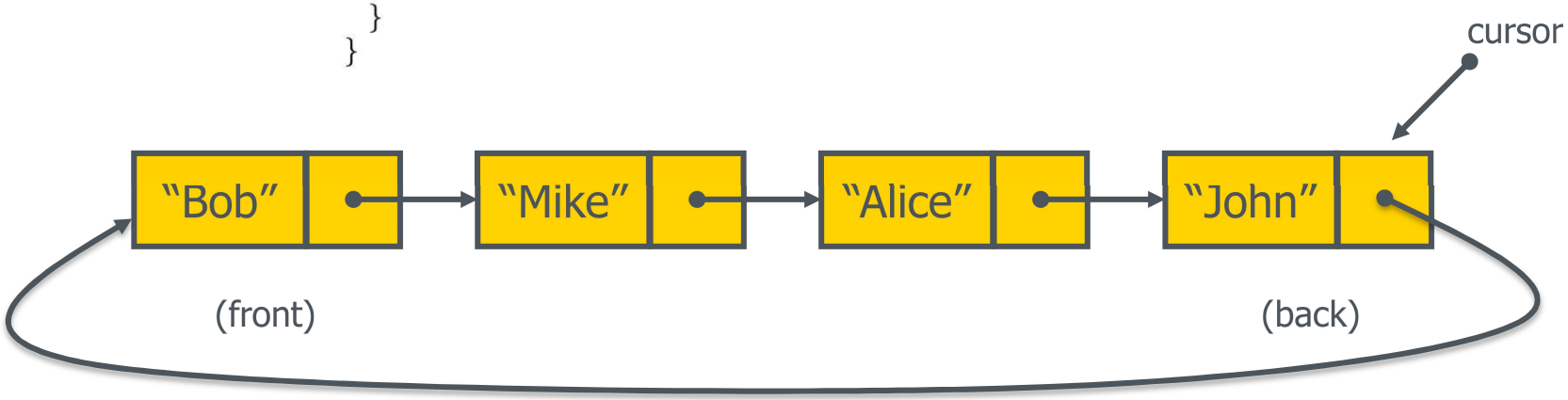
```
CircleList::CircleList()                    // constructor
  : cursor(NULL) { }
CircleList::~CircleList()                   // destructor
  { while (!empty()) remove(); }

bool CircleList::empty() const              // is list empty?
  { return cursor == NULL; }
const Elem& CircleList::back() const        // element at cursor
  { return cursor->elem; }
const Elem& CircleList::front() const       // element following cursor
  { return cursor->next->elem; }
void CircleList::advance()                  // advance cursor
  { cursor = cursor->next; }
```



Circular Linked List - add

- add an element
  - Insert a new node with element **e** immediately after the cursor; if the list is empty, then this node becomes the cursor and its next pointer points to itself.

```
void CircleList::add(const Elem& e) {      // add after cursor
  CNode* v = new CNode;                    // create a new node
  v->elem = e;
  if (cursor == NULL) {                    // list is empty?
    v->next = v;                           // v points to itself
    cursor = v;                            // cursor points to v
  }
  else {                                   // list is nonempty?
    v->next = cursor->next;                // link in v after cursor
    cursor->next = v;
  }
}
```

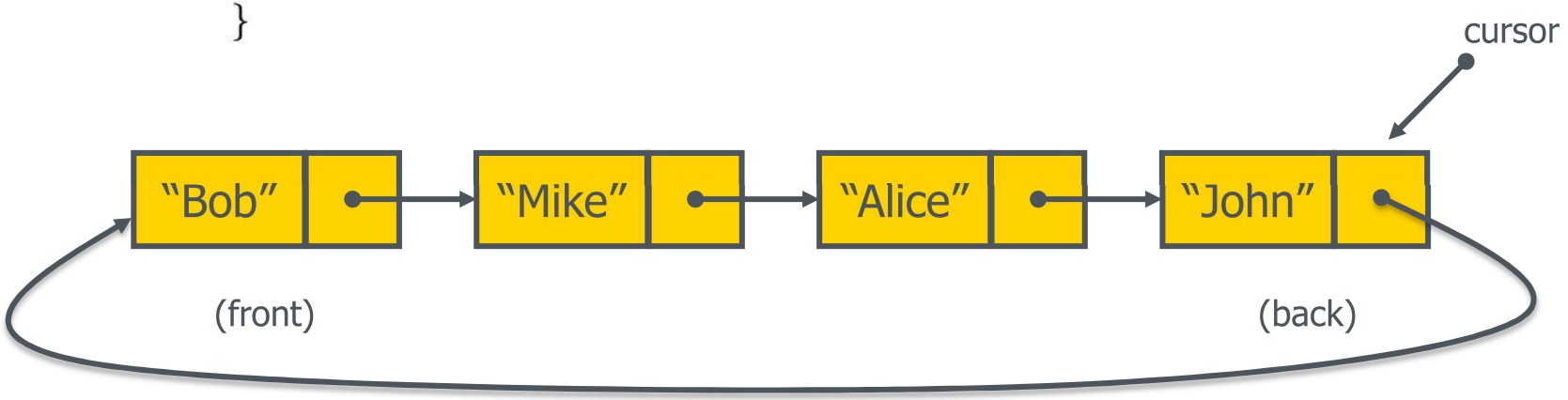


Circular Linked List - remove

- remove the element after cursor
  - Remove the node immediately after the cursor (not the cursor itself, unless it is the only node); if the list becomes empty, the cursor is set to null.

```
void CircleList::remove() {
  CNode* old = cursor->next;
  if (old == cursor)
    cursor = NULL;
  else
    cursor->next = old->next;
  delete old;
}
```

```
// remove node after cursor
// the node being removed
// removing the only node?
// list is now empty

// link out the old node
// delete the old node
```



Circularly Linked List - An Example

- A music playlist!

```
int main() {
  CircleList playList;                  // []
  playList.add("Stayin Alive");         // [Stayin Alive*]
  playList.add("Le Freak");             // [Le Freak, Stayin Alive*]
  playList.add("Jive Talkin");          // [Jive Talkin, Le Freak, Stayin Alive*]

  playList.advance();                   // [Le Freak, Stayin Alive, Jive Talkin*]
  playList.advance();                   // [Stayin Alive, Jive Talkin, Le Freak*]
  playList.remove();                    // [Jive Talkin, Le Freak*]
  playList.add("Disco Inferno");        // [Disco Inferno, Jive Talkin, Le Freak*]
  return EXIT_SUCCESS;
}
```

Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

13 Recursion

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

February 10, 2022

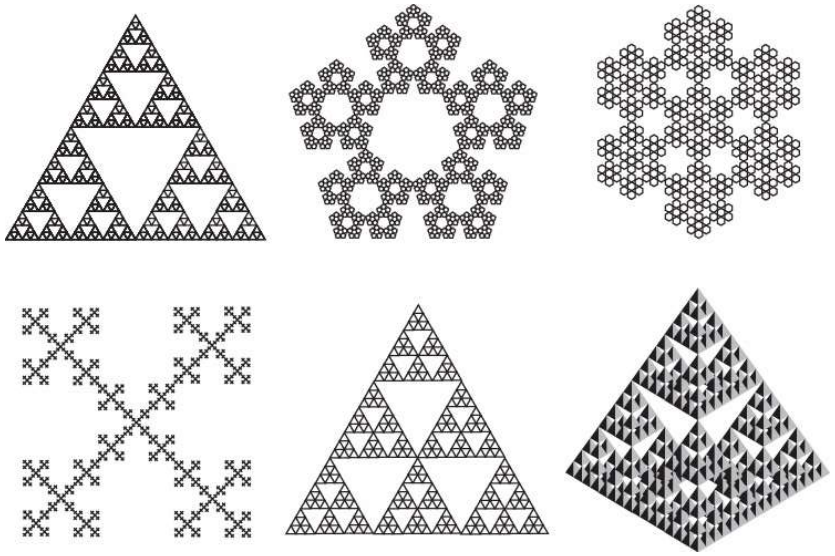
Administration

- Today's lecture is shorter
  - I will stay until 2:30 to answer questions of students who can not attend my office hour.

What is Recursion

- **Recursion** is the concept of defining a function that makes a call to itself.
- We call a function **Recursive** if it calls itself.
- When a function calls itself, we refer to this as a **Recursive Call**.
  - if function M calls another function that ultimately leads to a call back to M, this is also a recursive call and function M is recursive.

- A lot of use-cases in real-life:
  - Nature : fractals
  - Mathematics: recursive functions



Recursive Algorithms

- The idea is to avoid loops!
- A recursive implementation can be significantly simpler and easier to understand than an iterative implementation.
- Types of Recursion:
  - Linear Recursion
  - Binary Recursion
  - Tail Recursion
  - Multiple Recursion

Recursive Algorithms - Example

- Factorial of a whole number n is defined as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

  - factorial(4) = 4*(3*2*1) = 4*factorial(3)
  - factorial(4) can be defined in terms of factorial(3)
- Recursive definition of the factorial function is:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

  - base case
    - are non-recursive
  - recursive case
    - no circularity (function terminates)

Recursive Algorithms - Example

- Iterative Factorial

```
int factorial_iter(int n){
    int factorial = 1;
    for (int i = 2; i <= n; i++){
        factorial = factorial * i;
    }
    return factorial;
}
```

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

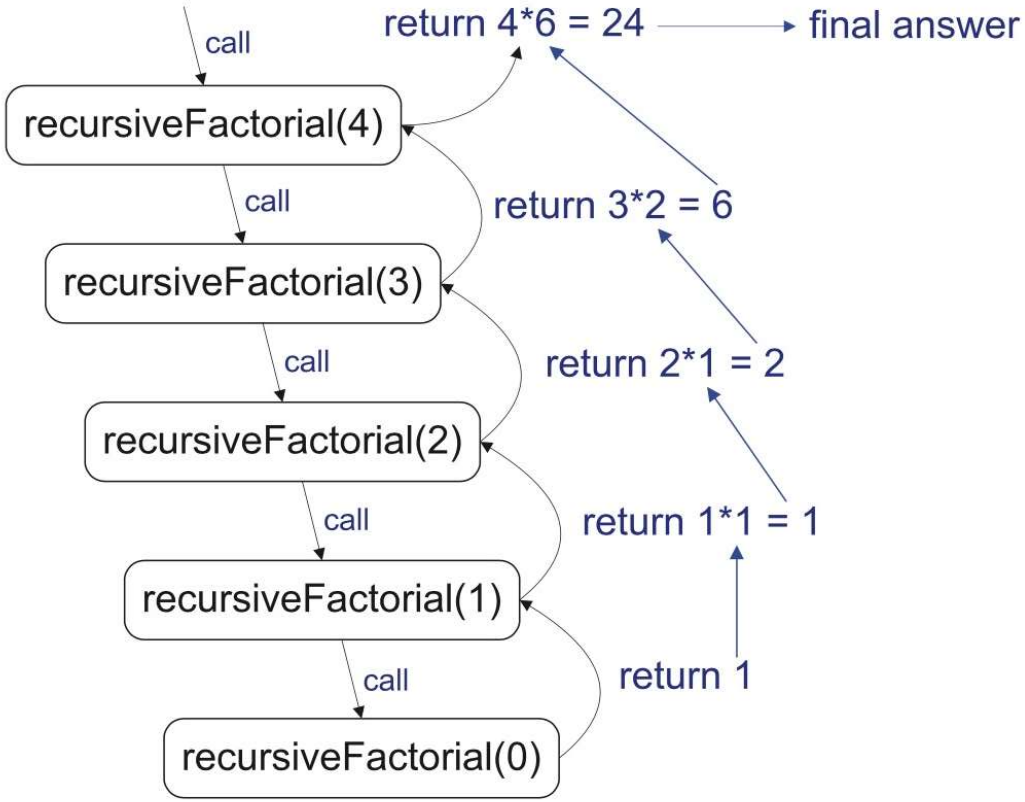
- Recursive Factorial:

```
int factorial_rec(int n){
    if (n==0){
        return 1;
    }
    else{
        return n * factorial_rec(n - 1);
    }
}
```

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

Recursive Algorithms - Example

- Recursive Factorial
- **Recursion Trace**:




```java
int factorial_rec(int n){
    if (n==0){
        return 1;
    }
    else{
        return n * factorial_rec(n - 1);
    }
}
```

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

Linear Recursion

- The simplest form of recursion
- function makes at most one recursive call each time it is invoked
- When we have a sequence, we can view some problems in terms of:
  - a first or last element
  - a remaining sequence that has the same structure as the original sequence

Linear Recursion

- The simplest form of recursion
- function makes at most one recursive call each time it is invoked
- When we have a sequence, we can view some problems in terms of:
  - a first or last element
  - a remaining sequence that has the same structure as the original sequence

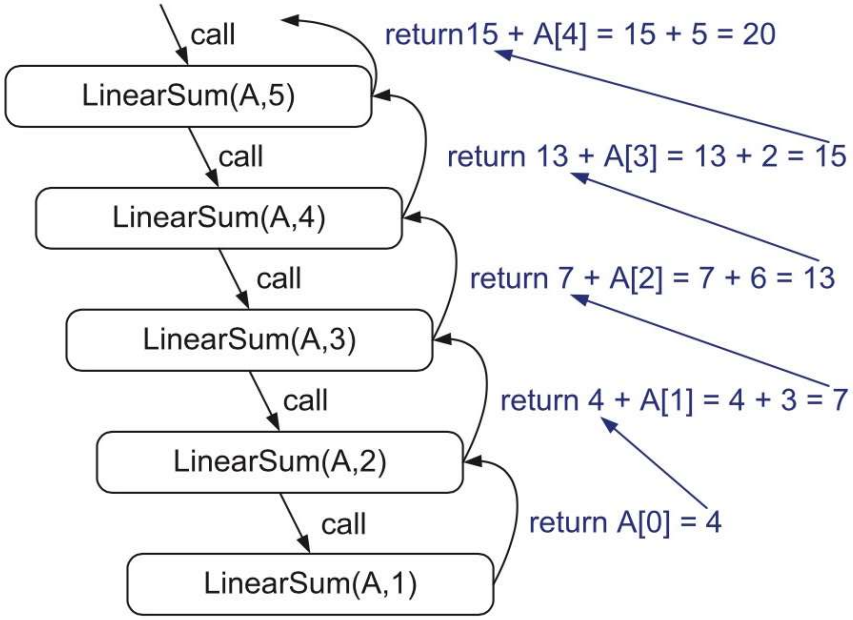
Algorithm LinearSum(A, n):
Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements
Output: The sum of the first n integers in A

```
if n = 1 then
    return A[0]
else
    return LinearSum(A, n − 1) + A[n − 1]
```

Linear Recursion

Algorithm LinearSum(A, n):
Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements
Output: The sum of the first n integers in A

if $n = 1$ **then**
  return $A[0]$
else
  return LinearSum$(A, n-1) + A[n-1]$




```
int linearSum_iter(int data[], int n){
    int sum = 0;
    for (int i =0; i < n; i++){
        sum += data[i];
    }
    return sum;
}
```

```
int linearSum(int data[], int n){
    if (n <= 0)
        return 0;
    return (linearSum(data, n - 1) + data[n - 1]);
}
```

Recursion in Computer Science

- Recursion is heavily used in Functional Programming
  - In Pure Functional Languages it is the only way to iterate!
    - Like Haskell
  - Non-pure programming languages allow iteration
    - Like Scala
- Recursion has overheads in Imperative Programming Languages
  - Like in C++ and Java
- Pure Functional Language use tail recursion conversions to reduce significant impact on memory consumption of heavy recursive calls
- It has many applications in Big Data Tools
  - Laziness!

Questions?

Those who can not attend the office hours can stay and ask questions for mid-term

If you don’t have questions, feel free to leave ☺

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

14 Recursion - continued

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

February 16, 2022

Administration

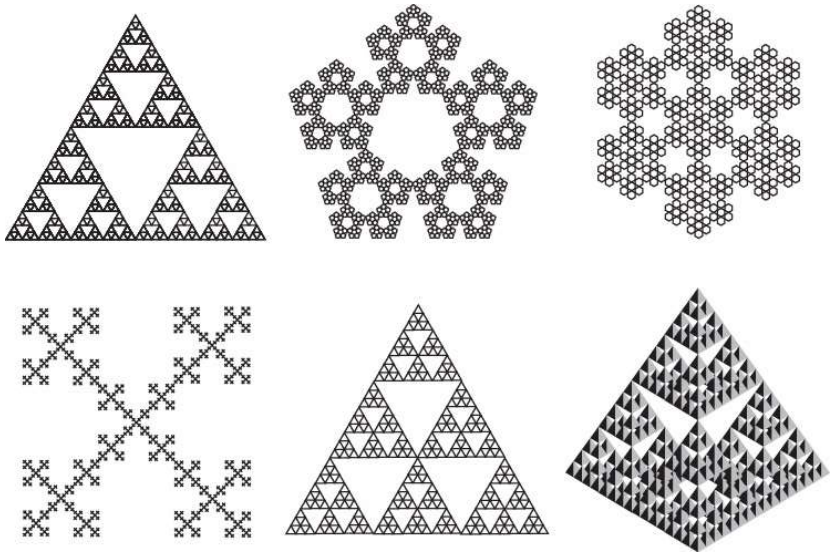
- I will return assignments today! finally!
  - Sorry about the delay
  - TAs did great, I delayed
- My Office Hour:
  - Today at 15:00 in ITB-159 in-person (or virtually using teams as usual)

- We review the recursion quickly

What is Recursion

- **Recursion** is the concept of defining a function that makes a call to itself.
- We call a function **Recursive** if it calls itself.
- When a function calls itself, we refer to this as a **Recursive Call**.
  - if function M calls another function that ultimately leads to a call back to M, this is also a recursive call and function M is recursive.

- A lot of use-cases in real-life:
  - Nature : fractals
  - Mathematics: recursive functions



Recursive Algorithms

- The idea is to avoid loops!
- A recursive implementation can be significantly simpler and easier to understand than an iterative implementation.
- Types of Recursion:
  - Linear Recursion
  - Binary Recursion
  - Tail Recursion
  - Multiple Recursion

Recursive Algorithms - Example

- Factorial of a whole number n is defined as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

  - factorial(4) = 4*(3*2*1) = 4*factorial(3)
  - factorial(4) can be defined in terms of factorial(3)
- Recursive definition of the factorial function is:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

  - base case
    - are non-recursive
  - recursive case
    - no circularity (function terminates)

Recursive Algorithms - Example

- Iterative Factorial

```
int factorial_iter(int n){
    int factorial = 1;
    for (int i = 2; i <= n; i++){
        factorial = factorial * i;
    }
    return factorial;
}
```

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

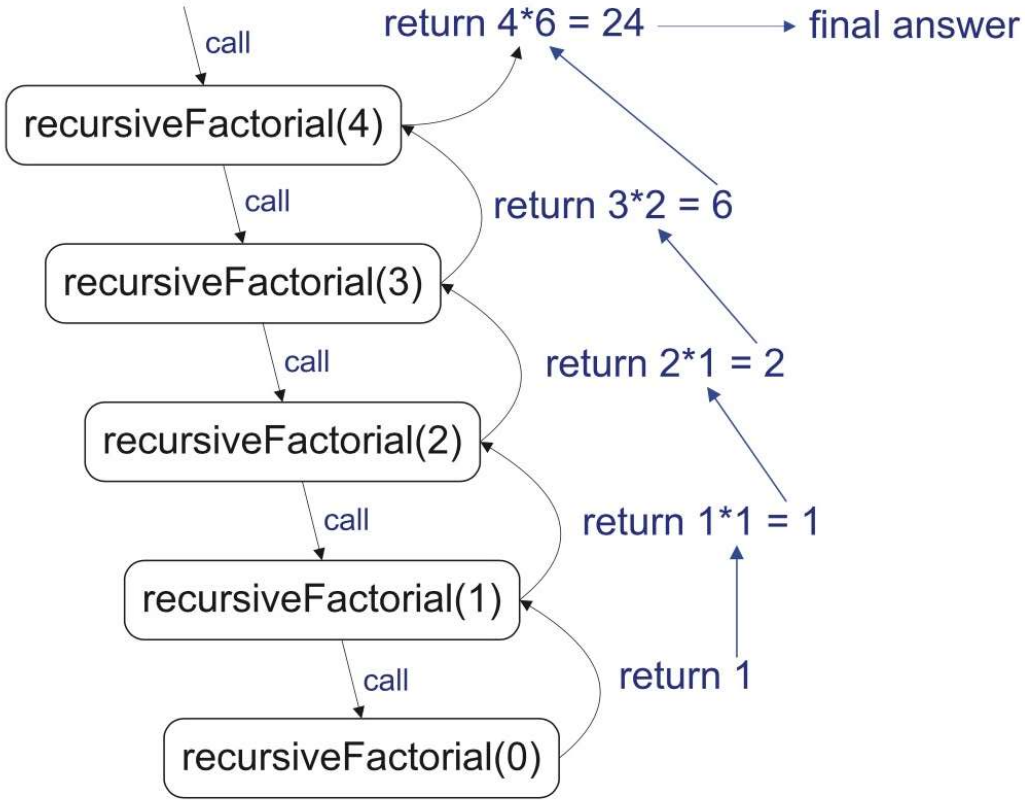
- Recursive Factorial:

```
int factorial_rec(int n){
    if (n==0){
        return 1;
    }
    else{
        return n * factorial_rec(n - 1);
    }
}
```

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

Recursive Algorithms - Example

- Recursive Factorial
- **Recursion Trace**:




```
int factorial_rec(int n){
    if (n==0){
        return 1;
    }
    else{
        return n * factorial_rec(n - 1);
    }
}
```

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

Linear Recursion

- The simplest form of recursion
- function makes at most one recursive call each time it is invoked
- When we have a sequence, we can view some problems in terms of:
  - a first or last element
  - a remaining sequence that has the same structure as the original sequence

Linear Recursion

- The simplest form of recursion
- function makes at most one recursive call each time it is invoked
- When we have a sequence, we can view some problems in terms of:
  - a first or last element
  - a remaining sequence that has the same structure as the original sequence

Algorithm LinearSum(A, n):
Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements
Output: The sum of the first n integers in A

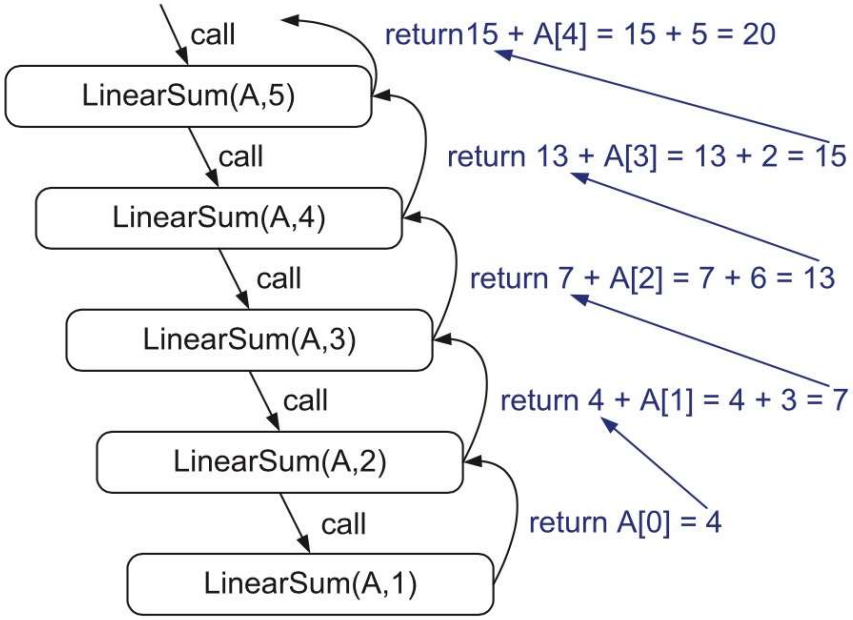
```
if n = 1 then
    return A[0]
else
    return LinearSum(A, n − 1) + A[n − 1]
```

Linear Recursion

Algorithm LinearSum(A, n):
Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements
Output: The sum of the first n integers in A

if $n = 1$ **then**
  return $A[0]$
else
  return LinearSum$(A, n-1) + A[n-1]$

$A = \{4,3,6,2,5\}$




```java
int linearSum_iter(int data[], int n){
    int sum = 0;
    for (int i =0; i < n; i++){
        sum += data[i];
    }
    return sum;
}
```

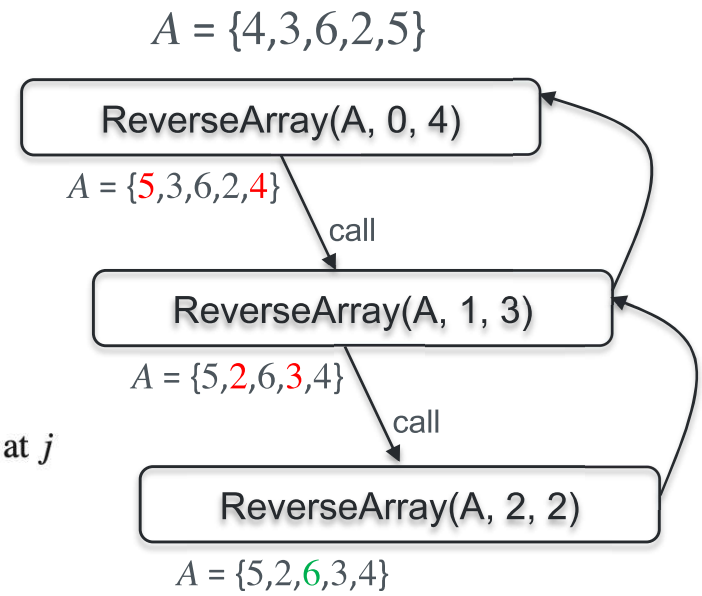
```java
int linearSum(int data[], int n){
    if (n <= 0)
        return 0;
    return (linearSum(data, n - 1) + data[n - 1]);
}
```

Recursion in Computer Science

- Recursion is heavily used in Functional Programming
  - In Pure Functional Languages it is the only way to iterate!
    - Like Haskell
  - Non-pure programming languages allow iteration
    - Like Scala
- Recursion has overheads in Imperative Programming Languages
  - Like in C++ and Java
- Pure Functional Language use tail recursion conversions to reduce significant impact on memory consumption of heavy recursive calls
- It has many applications in Big Data Tools
  - Laziness!

Linear Recursion - Reversing an Array

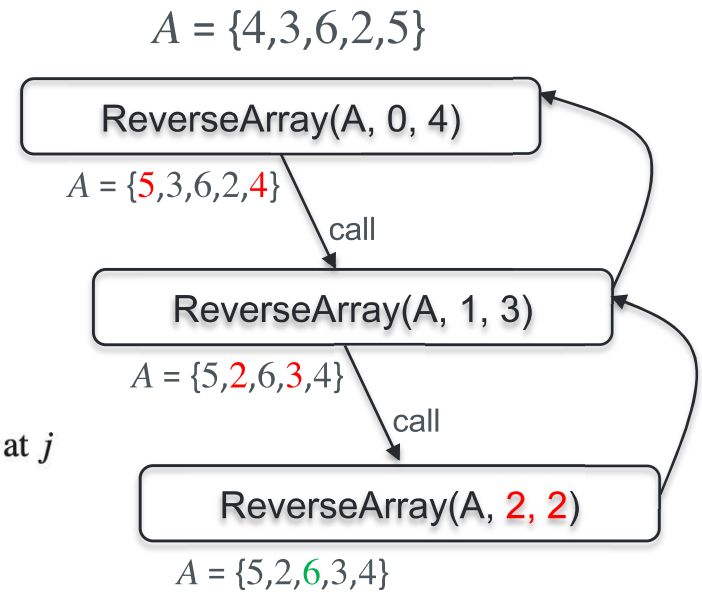
- Reversing the n elements of an array
- Reversal: by swapping the first and last elements and then recursively reversing the remaining elements in the array.
  - Note how subproblems are defined! (initial call ReverseArray(A, 0, n-1)
    - other than array, we have two other index inputs
- Base cases?



Algorithm ReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j
if $i < j$ **then**
  Swap $A[i]$ and $A[j]$
  ReverseArray$(A, i+1, j-1)$
return

Linear Recursion - Reversing an Array

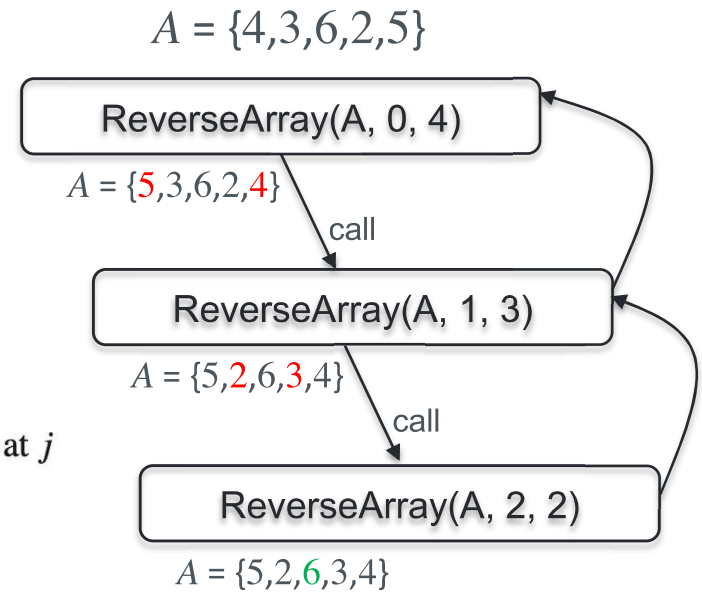
- Reversing the n elements of an array
- Reversal: by swapping the first and last elements and then recursively reversing the remaining elements in the array.
  - Note how subproblems are defined! (initial call ReverseArray(A, 0, n-1)
    - other than array, we have two other index inputs
- Base cases?
  - if **i = j** or **i > j**
    - in either of cases algorithm terminates



Algorithm ReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j
if $i < j$ **then**
  Swap $A[i]$ and $A[j]$
  ReverseArray$(A, i+1, j-1)$
return

Linear Recursion - Reversing an Array

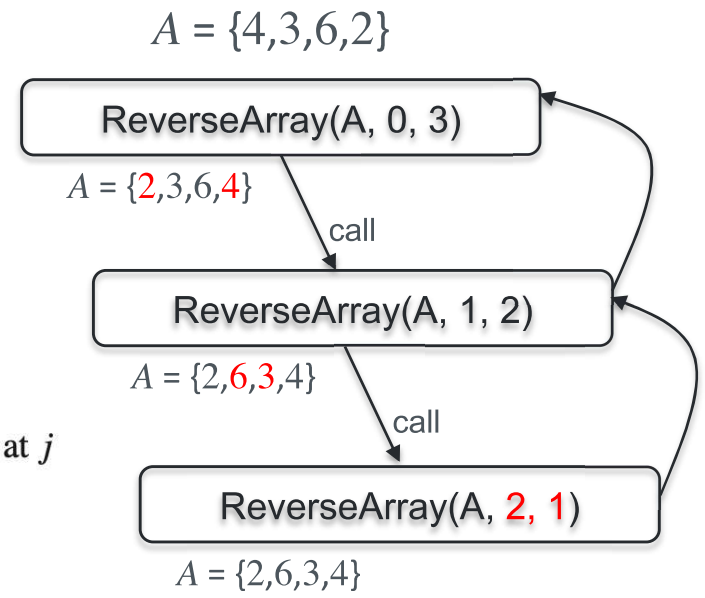
- Reversing the n elements of an array
- Reversal: by swapping the first and last elements and then recursively reversing the remaining elements in the array.
  - Note how subproblems are defined! (initial call ReverseArray(A, 0, n-1)
    - other than array, we have two other index inputs
- Base cases?
  - if **i = j** or **i > j**
    - in either of cases algorithm terminates
- Does algorithm terminate?



Algorithm ReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j
if $i < j$ **then**
  Swap $A[i]$ and $A[j]$
  ReverseArray$(A, i+1, j-1)$
return

Linear Recursion - Reversing an Array

- Reversing the n elements of an array
- Reversal: by swapping the first and last elements and then recursively reversing the remaining elements in the array.
  - Note how subproblems are defined! (initial call ReverseArray(A, 0, n-1)
    - other than array, we have two other index inputs
- Base cases?
  - if **i = j** or **i > j**
    - in either of cases algorithm terminates
- Does algorithm terminate?



Algorithm ReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j
if $i < j$ **then**
  Swap $A[i]$ and $A[j]$
  ReverseArray$(A, i+1, j-1)$
return

Tail Recursion

- Recursive calls are costly
  - Stack memory keeps track of the state of each active recursive call
- We can convert some recursive algorithms into non-recursive version
  - Suitable for imperative languages like C++
- There are two conditions:
  - The recursion is Linear
  - The recursive call is the last thing that function does

Algorithm ReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j
if $i < j$ **then**
  Swap $A[i]$ and $A[j]$
  ReverseArray$(A, i+1, j-1)$ Recursive call is the last thing!
return

Tail Recursion

- Recursive calls are costly
  - Stack memory keeps track of the state of each active recursive call
- We can convert some recursive algorithms into non-recursive version
  - Suitable for imperative languages like C++
- There are two conditions:
  - The recursion is Linear
  - The recursive call is the last thing that function does

Algorithm LinearSum(A,n):
Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements
Output: The sum of the first n integers in A

```
if n = 1 then
    return A[0]
else
    return LinearSum(A, n − 1) + A[n − 1]
```

Recursive call is NOT the last thing!

Tail Recursion

- There are two conditions:
  - The recursion is Linear
  - The recursive call is the last thing that function does
- **Iterate** through the recursive calls rather than calling them explicitly

- Linear Recursive version:

Algorithm ReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j
if $i < j$ **then**
  Swap $A[i]$ and $A[j]$
  ReverseArray$(A, i+1, j-1)$
return

- Tail recursion conversion:

  (iterative)

Algorithm IterativeReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j
while $i < j$ **do**
  Swap $A[i]$ and $A[j]$
  $i \leftarrow i+1$
  $j \leftarrow j-1$
return

Binary Recursion

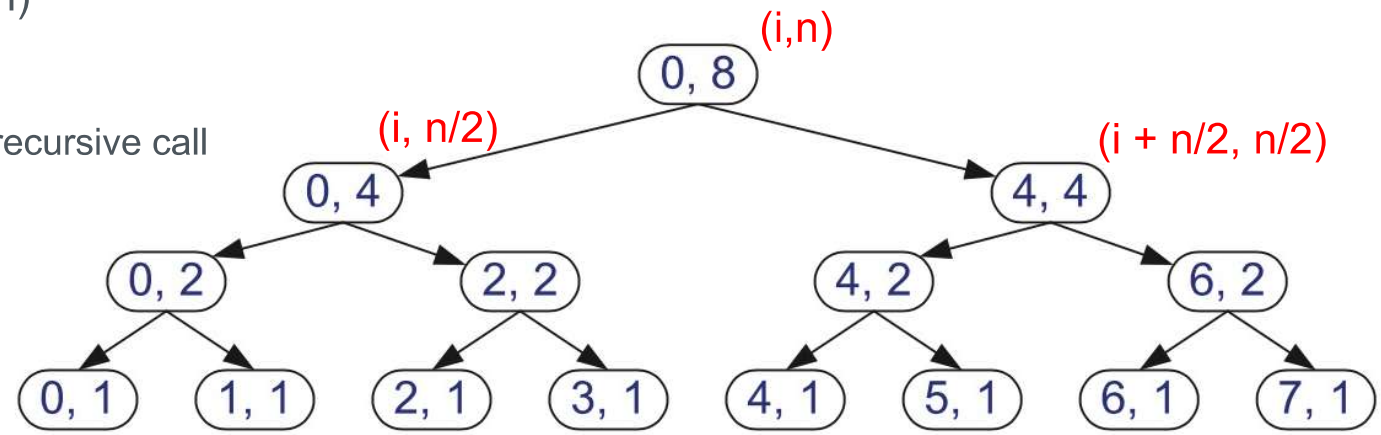
- When an algorithm makes **two** recursive calls
  - Useful to solve two similar halves of a problem
- Again! summing the **n** elements of an integer array A:
  - recursively
    - summing first half
    - summing second half
    - adding the two

Algorithm BinarySum(A, i, n):
Input: An array A and integers i and n
Output: The sum of the n integers in A starting at index i
if $n = 1$ **then**
return $A[i]$
return BinarySum$(A, i, \lceil n/2 \rceil)$ + BinarySum$(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

Binary Recursion

- Again! summing the **n** elements of an integer array **A**:
  - recursively
    - summing first half
    - summing second half
    - adding the two
- Analysis of the algorithm:
  - We assume n is a power of 2
  - BinarySum(A, 0, n)
    - BinarySum(A, 0, 8)
  - *n* is halved at each recursive call

Algorithm BinarySum(A, i, n):
Input: An array A and integers i and n
Output: The sum of the n integers in A starting at index i
if $n = 1$ **then**
return $A[i]$
return BinarySum$(A, i, \lceil n/2 \rceil)$ + BinarySum$(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

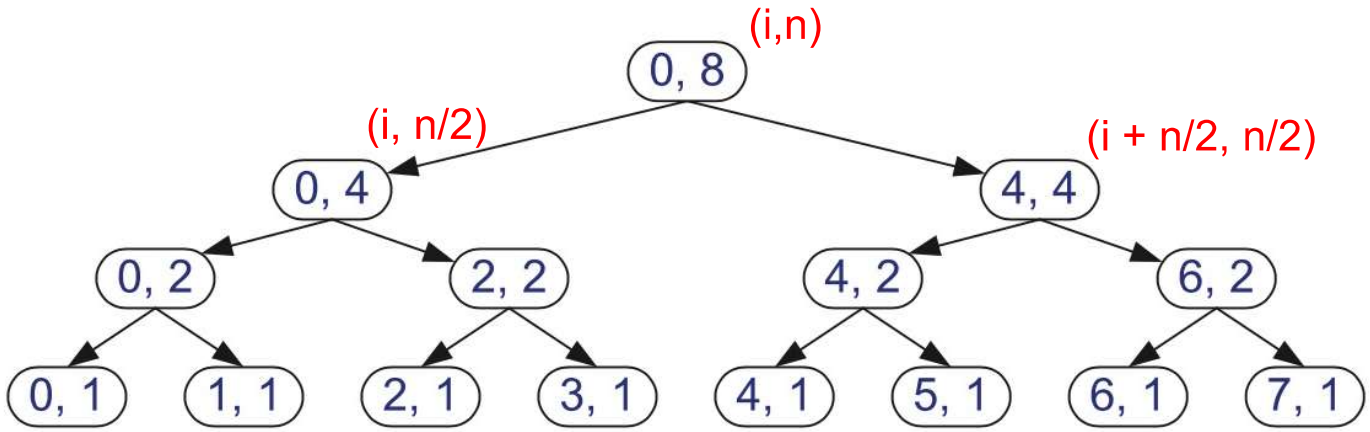


Binary Recursion

- Analysis of the algorithm:
  - We assume n is a power of 2
  - BinarySum(A, 0, n)
    - BinarySum(A, 0, 8)
  - *n* is halved at each recursive call

Algorithm BinarySum(A, i, n):
Input: An array A and integers i and n
Output: The sum of the n integers in A starting at index i
if $n = 1$ **then**
return $A[i]$
return BinarySum$(A, i, \lceil n/2 \rceil)$ + BinarySum$(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

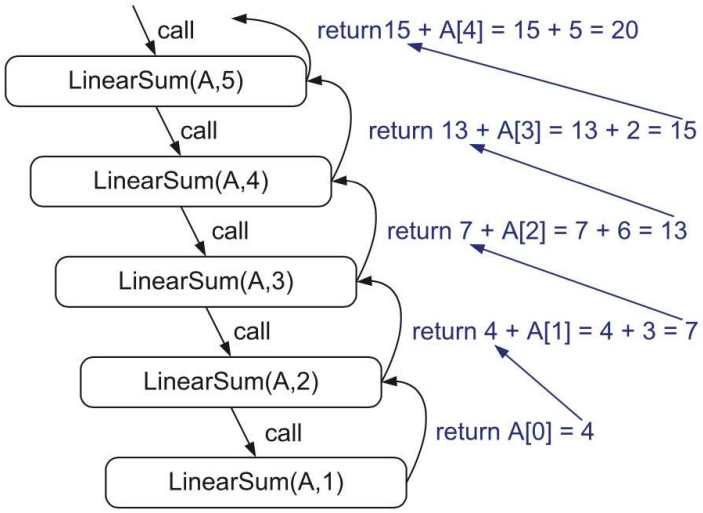
- The depth of the recursion, that is, the maximum number of function instances that are active at the same time, is **1 + $\log_2 n$**



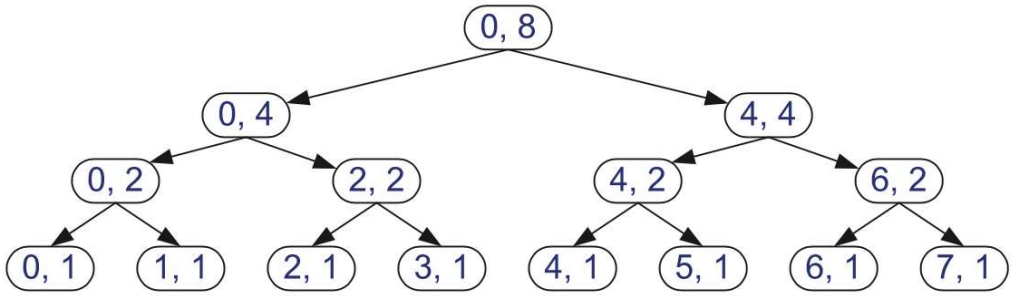
Binary Recursion

- Analysis of the algorithm:
  - We assume n is a power of 2
- **Memory consumption** (Space complexity)
  - The depth of the recursion, that is, the maximum number of function instances that are active at the same time, is **1 + $\log_2 n$**
  - Remember that this depth was **n** for **LinearSum**

LinearSum(A, 5)



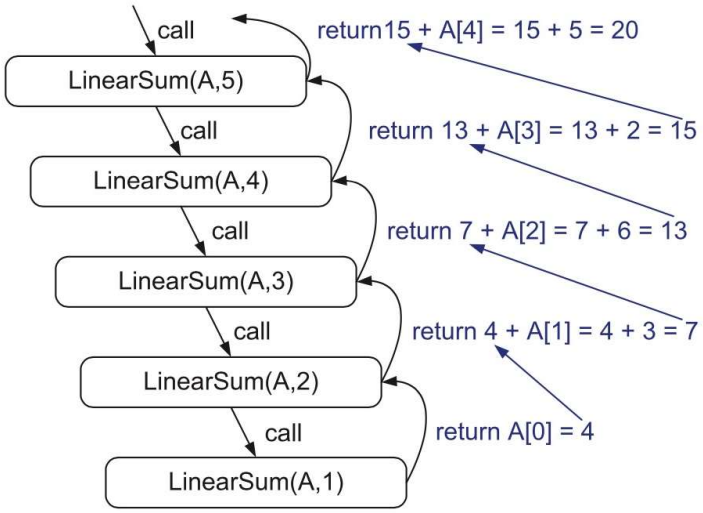
BinarySum(A, 0, 8)



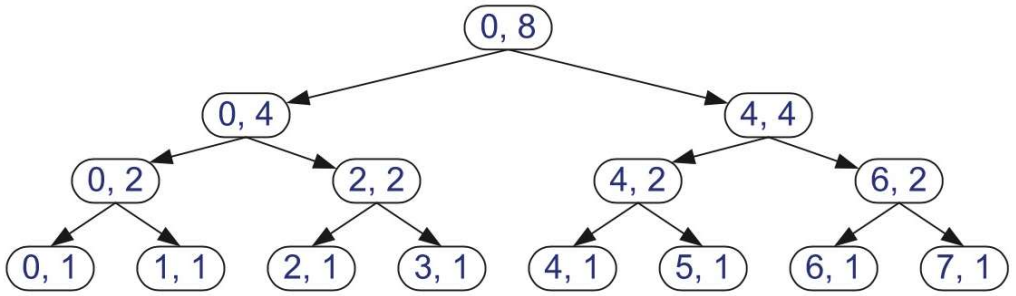
Binary Recursion

- Analysis of the algorithm:
  - We assume n is a power of 2
- **Runtime** (Time complexity)
  - There are **2n-1** boxes (calls) in **BinarySum**
  - There are **n** boxes (calls) in **LinearSum**
- Assume each calls is visited in a constant time

LinearSum(A, 5)



BinarySum(A, 0, 8)



Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

15 Algorithms Analysis

Department of Computing and Software

Instructor:

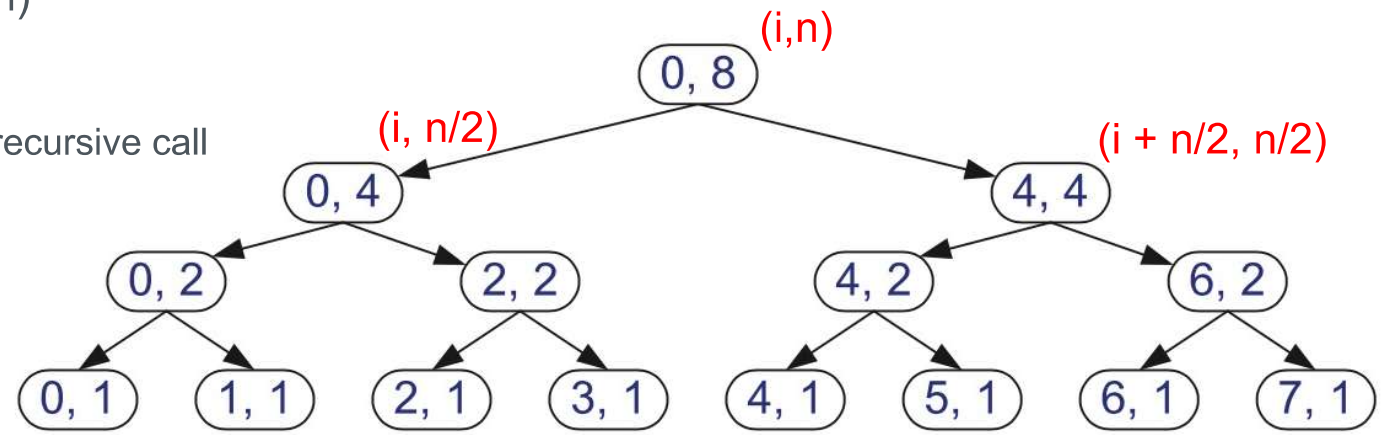
Omid Isfahanialamdari

February 17, 2022

Binary Recursion (from last lecture)

- Again! summing the **n** elements of an integer array **A**:
  - recursively
    - summing first half
    - summing second half
    - adding the two
- Analysis of the algorithm:
  - We assume n is a power of 2
  - BinarySum(A, 0, n)
    - BinarySum(A, 0, 8)
  - *n* is halved at each recursive call

Algorithm BinarySum(A, i, n):
Input: An array A and integers i and n
Output: The sum of the n integers in A starting at index i
if $n = 1$ **then**
return $A[i]$
return BinarySum$(A, i, \lceil n/2 \rceil)$ + BinarySum$(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

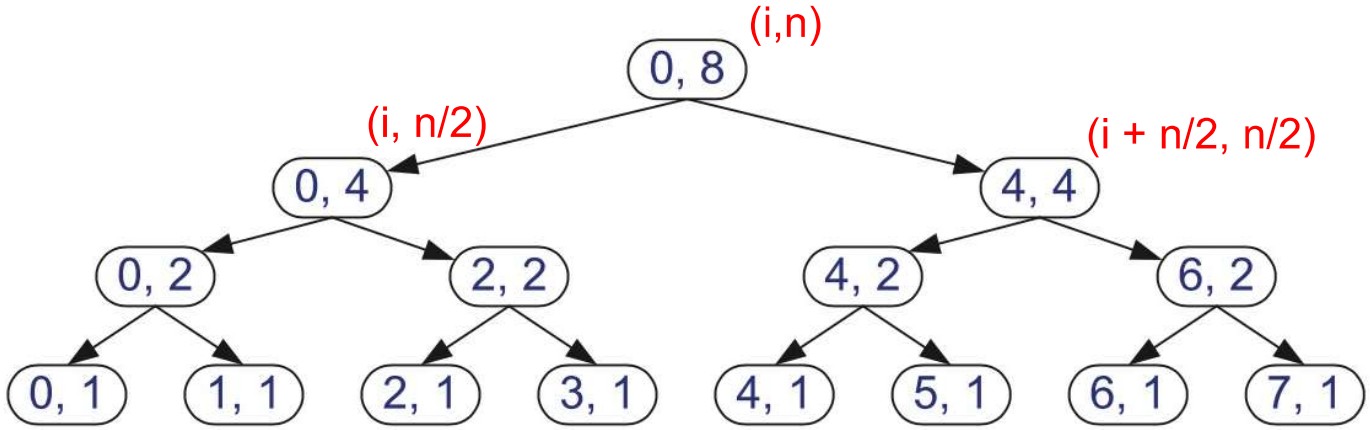


Binary Recursion (from last lecture)

- Analysis of the algorithm:
  - We assume n is a power of 2
  - BinarySum(A, 0, n)
    - BinarySum(A, 0, 8)
  - *n* is halved at each recursive call

Algorithm BinarySum(A, i, n):
Input: An array A and integers i and n
Output: The sum of the n integers in A starting at index i
if $n = 1$ **then**
  return $A[i]$
return BinarySum$(A, i, \lceil n/2 \rceil)$ + BinarySum$(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

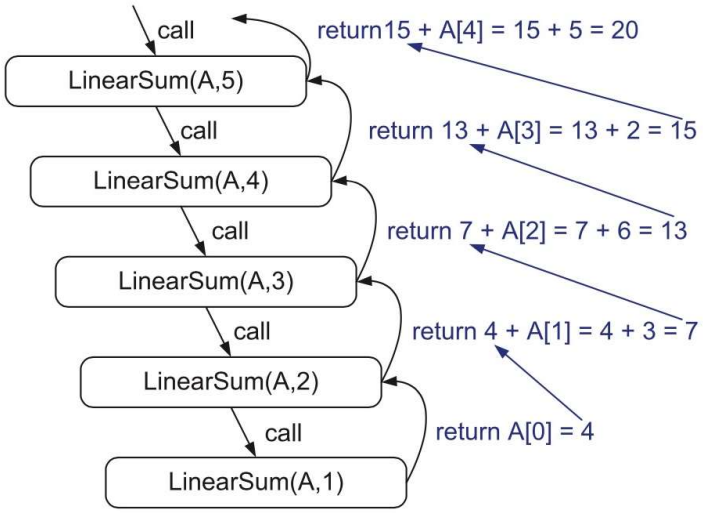
- The depth of the recursion, that is, the maximum number of function instances that are active at the same time, is **1 + $\log_2 n$**



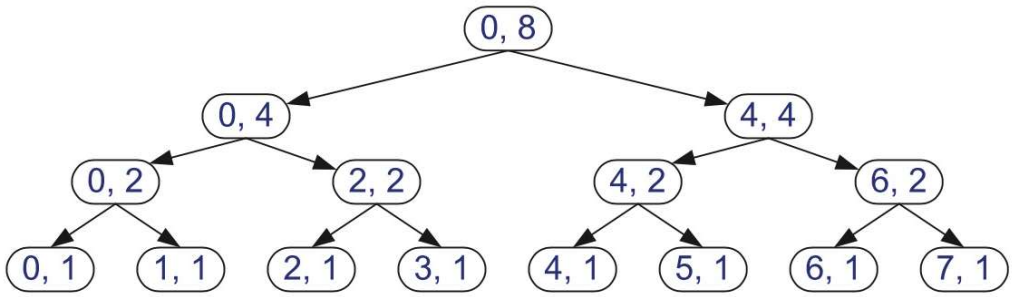
Binary Recursion (from last lecture)

- Analysis of the algorithm:
  - We assume n is a power of 2
- **Memory consumption** (Space complexity)
  - The depth of the recursion, that is, the maximum number of function instances that are active at the same time, is **1 + $\log_2 n$**
  - Remember that this depth was **n** for **LinearSum**

LinearSum(A, 5)



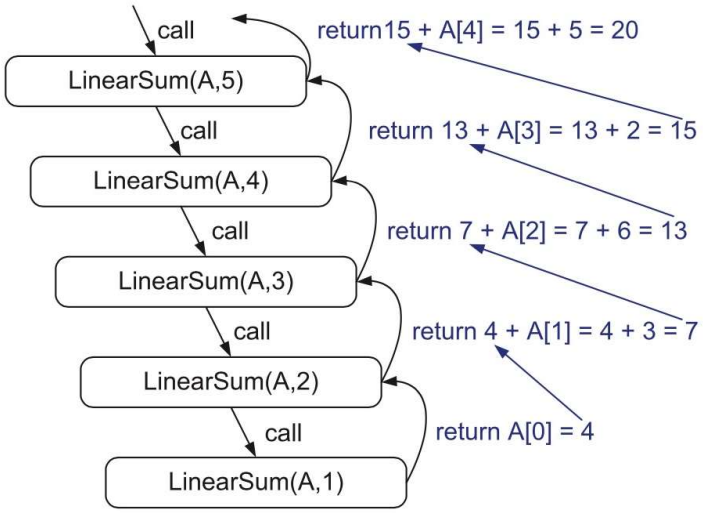
BinarySum(A, 0, 8)



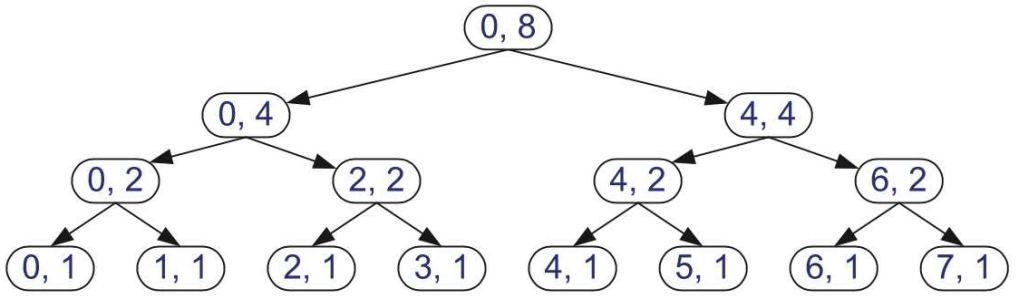
Binary Recursion (from last lecture)

- Analysis of the algorithm:
  - We assume n is a power of 2
- **Runtime** (Time complexity)
  - There are **2n-1** boxes (calls) in **BinarySum**
  - There are **n** boxes (calls) in **LinearSum**
- Assume each calls is visited in a constant time

LinearSum(A, 5)



BinarySum(A, 0, 8)

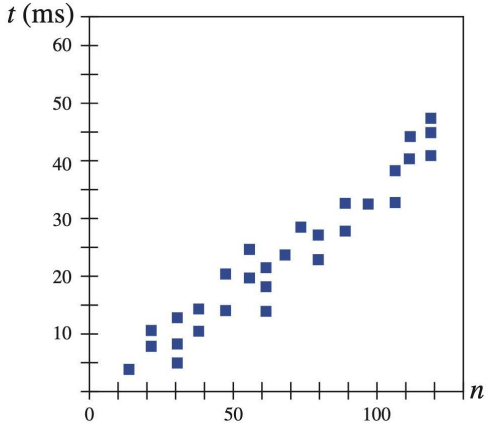


Comparing Algorithms

- Algorithm: An algorithm is a step-by-step procedure for solving a problem in a finite amount of time.
- There are different solutions for a problem
- We need to identify. which solution is **better** than the other.
- Better in the sense of:
  - Running time
  - Memory space required
  - structure of programs (readability, simplicity, design)

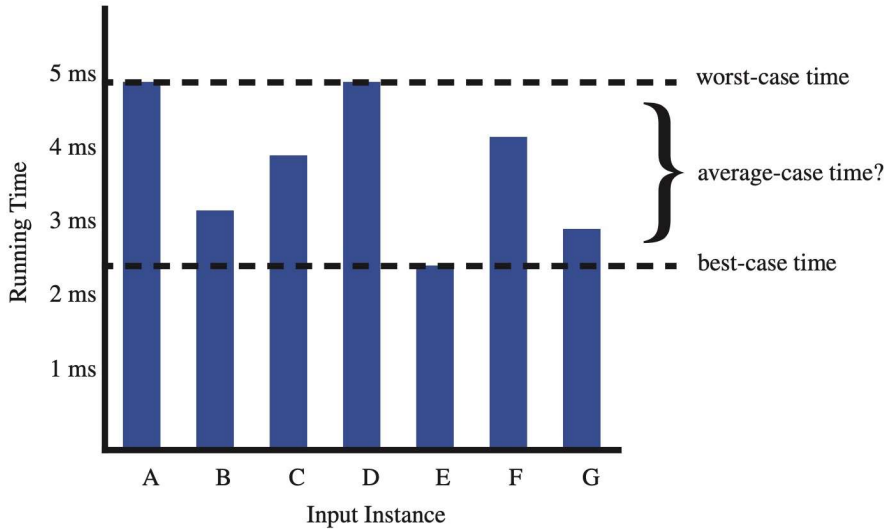
Running Time

- Most algorithms transform input objects into output objects.
  - We need to think about input
    - Insertion sort: array of size **n**
    - recursive factorial: **n**
- The running time of an algorithm typically grows with the input size.
  - best cases
  - average cases
  - worst cases
- We focus on the worst-case running time.
  - Easier to analyze
  - Crucial to applications such as games, embedded systems
- Average-case running time is often difficult to determine.
  - Why?



Average vs. Worst Case

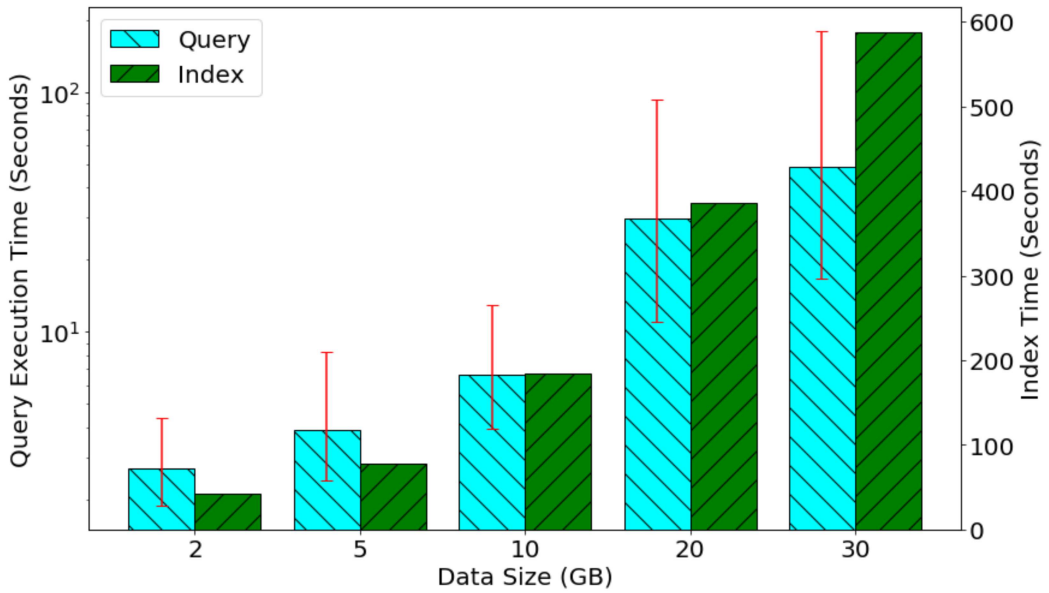
- The average case running time is harder to analyze because you need to know the probability distribution of the input.



- knowing the worst-case time is important in many applications
  - real-time systems
  - nuclear reactor
  - air traffic control

Experimental Approach

- Implement your algorithm
- Run the program with inputs of varying size and composition
- Use a wall clock to get an accurate measure of the actual running time
- Plot the results



Experimental Approach - Limitations

- It is necessary to implement your algorithm, which may be difficult and often time-consuming
  - Sometimes you just ideate!
- Results may not be indicative of the running time on other inputs not included in the experiment.
  - Your input may not be inclusive enough of all possible inputs
- Competing algorithms!
  - In order to compare your algorithm with a competing algorithm, the same hardware and software environments must be used
    - Your competitor may have ran on a high-end machine to which you don't have access
      - You need to implement it!
      - Or buy the same machine

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
  - The algorithm receives a worth of data
  - Performs a few operations on the data
- Characterizes running time as a function of the input
  - Size of input: array of length **n**
  - input: **n** for factorial
- Considers all possible inputs
- Allows us to evaluate the **relative efficiency** of any two algorithms **independent of** the hardware/software environment
- For each algorithm, we will end up with a function **f(n)** that characterizes the running time of the algorithm as a function of the input size **n**

Primitive Operations

- Primitive operation corresponds to a low-level instruction with an execution time that is constant
  - Basic computations performed by an algorithm
  - Identifiable in pseudocode
  - Largely independent of the programming language
  - Exact definition is not important
- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.

```
currMax ← A[0]
for i ← 1 to n − 1 do
    if currMax < A[i] then
        currMax ← A[i]
return currMax
```

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.

$currMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.
  $currMax \leftarrow A[0]$ <-------- 2 operations
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $currMax < A[i]$ **then**
      $currMax \leftarrow A[i]$
  return $currMax$

1. accessing A[0] (indexing in array)
2. assigning A[0] to currMax

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

The for loop repeats **n** times, why?
Each time it has **2** operations, why?

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

The for loop repeats **n** times, why?
Each time it has **2** operations, why?
each time it involves an **assignment** and a **comparison**

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

```
currMax ← A[0]          <-------- 2 operations
for i ← 1 to n − 1 do
    if currMax < A[i] then
        currMax ← A[i]
return currMax
```

The for loop will repeat **n** times, why?

We account for the last increment to **n** in which the for loop identifies it should exit before entering next iteration
for example: n = 4
for ***i*** from ***1*** to ***3*** ➔ in the last iteration ***i*** becomes ***4*** and will be compared to ***3***

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

```
currMax ← A[0]                    <-------- 2 operations
for i ← 1 to n − 1 do             <----- 2n operations
    if currMax < A[i] then
        currMax ← A[i]
return currMax
```

The for loop will repeat **n** times, why?

We account for the last increment to **n** in which the for loop identifies it should exit before entering next iteration
for example: n = 4
for ***i*** from ***1*** to ***3*** ➔ in the last iteration ***i*** becomes ***4*** and will be compared to ***3***

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations
for $i \leftarrow 1$ **to** $n-1$ **do** <----- 2n operations
if $currMax < A[i]$ **then** <-- 2(n-1) operations
$currMax \leftarrow A[i]$ <-- 2(n-1) operations
i++ <-- 2(n-1) operations
return $currMax$

The body of for loop will repeat **n-1** times
The increment of ***i*** is performed at the end of each iteration

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

for $i \leftarrow 1$ **to** $n - 1$ **do** <----- 2n operations

if $currMax < A[i]$ **then** <-- 2(n-1) operations

$currMax \leftarrow A[i]$ <-- 2(n-1) operations

i++ <-- 2(n-1) operations

return $currMax$ <-- 1 operation

We will have a total of 8n - 2 operations

n is the input size!

Estimating Runtime

- Algorithm **arrayMax** executes 8n - 2 primitive operations in total

```
Algorithm arrayMax(A, n):
    Input: An array A storing n ≥ 1 integers.
    Output: The maximum element in A.
currMax ← A[0]
for i ← 1 to n − 1 do
    if currMax < A[i] then
        currMax ← A[i]
return currMax
```

- Suppose:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- The running time of **arrayMax** is bounded by two linear functions
  - **a**(8n - 2) <= **T(n)** <= **b**(8n - 2)
- Changing the hardware / software environment
  - Affects **T(n)** by a constant factor, but does not alter the growth rate of T(n)
- The linear growth rate of the running time **T(n)** is an **intrinsic property** of algorithm **arrayMax**

Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

16 Algorithms Analysis (cont.)

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

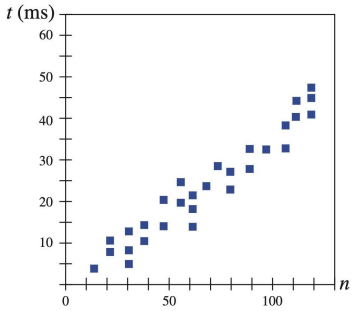
February 28, 2022

Administration

- Please pay special attention to this week's tutorial
- Start the assignment as early as possible. I will not be able to answer questions during coming weekend!
  - If you have questions, ask early.

Review

- Runtime Complexity Analysis
  - Experimental approach
    - Implement, run with varying input sizes, and measure run-times



  - Theoretical Approach
    - Allows us to evaluate the **relative efficiency** of any two algorithms **independent of** the hardware/software environment
    - For each algorithm, we will end up with a function **f(n)** that characterizes the running time of the algorithm as a function of the input size **n.**
    - We started by looking at primitive operations to get an idea of what operations an algorithm perform on input

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.

$currMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.
  $currMax \leftarrow A[0]$ <-------- 2 operations
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $currMax < A[i]$ **then**
      $currMax \leftarrow A[i]$
  return $currMax$

1. accessing A[0] (indexing in array)
2. assigning A[0] to currMax

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

The for loop repeats **n** times, why?
Each time it has **2** operations, why?

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

The for loop repeats **n** times, why?
Each time it has **2** operations, why?
each time it involves an **assignment** and a **comparison**

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

The for loop will repeat **n** times, why?

We account for the last increment to **n** in which the for loop identifies it should exit before entering next iteration
for example: n = 4
for ***i*** from ***1*** to ***3*** ➔ in the last iteration ***i*** becomes ***4*** and will be compared to ***3***

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

```
currMax ← A[0]                    <-------- 2 operations
for i ← 1 to n − 1 do             <----- 2n operations
    if currMax < A[i] then
        currMax ← A[i]
return currMax
```

The for loop will repeat **n** times, why?

We account for the last increment to **n** in which the for loop identifies it should exit before entering next iteration
for example: n = 4
for ***i*** from ***1*** to ***3*** ➔ in the last iteration ***i*** becomes ***4*** and will be compared to ***3***

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations
for $i \leftarrow 1$ **to** $n-1$ **do** <----- 2n operations
  if $currMax < A[i]$ **then** <-- 2(n-1) operations
    $currMax \leftarrow A[i]$ <-- 2(n-1) operations
  i++ <-- 2(n-1) operations
return $currMax$

The body of for loop will repeat **n-1** times
The increment of ***i*** is performed at the end of each iteration

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

for $i \leftarrow 1$ **to** $n-1$ **do** <----- 2n operations

if $currMax < A[i]$ **then** <-- 2(n-1) operations

$currMax \leftarrow A[i]$ <-- 2(n-1) operations

i++ <-- 2(n-1) operations

return $currMax$ <-- 1 operation

We will have a total of 8n - 3 operations

n is the input size!

Estimating Runtime

- Algorithm **arrayMax** executes 8n - 3 primitive operations in total

```
Algorithm arrayMax(A, n):
    Input: An array A storing n ≥ 1 integers.
    Output: The maximum element in A.
currMax ← A[0]
for i ← 1 to n − 1 do
    if currMax < A[i] then
        currMax ← A[i]
return currMax
```

- Suppose:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- The running time of **arrayMax** is bounded by two linear functions
  - **a**(8n - 3) <= **T(n)** <= **b**(8n - 3)
- Changing the hardware / software environment
  - Affects **T(n)** by a constant factor, but does not alter the growth rate of T(n)
- The linear growth rate of the running time **T(n)** is an **intrinsic property** of algorithm **arrayMax**

Asymptotic Notation

- Big-picture approach
  - In algorithm analysis, we focus on the growth rate of the running time as a function of the input size **n**.
  - It is often enough just to know that the running time of an algorithm such as arrayMax, grows proportionally to **n**, with its true running time being **n** times a **constant factor** that depends on the specific computer. (was 8n - 3)
  - We characterize the running times of algorithms by using functions that map the size of the input, **n**, to values that correspond to the main factor that determines the growth rate in terms of **n**.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

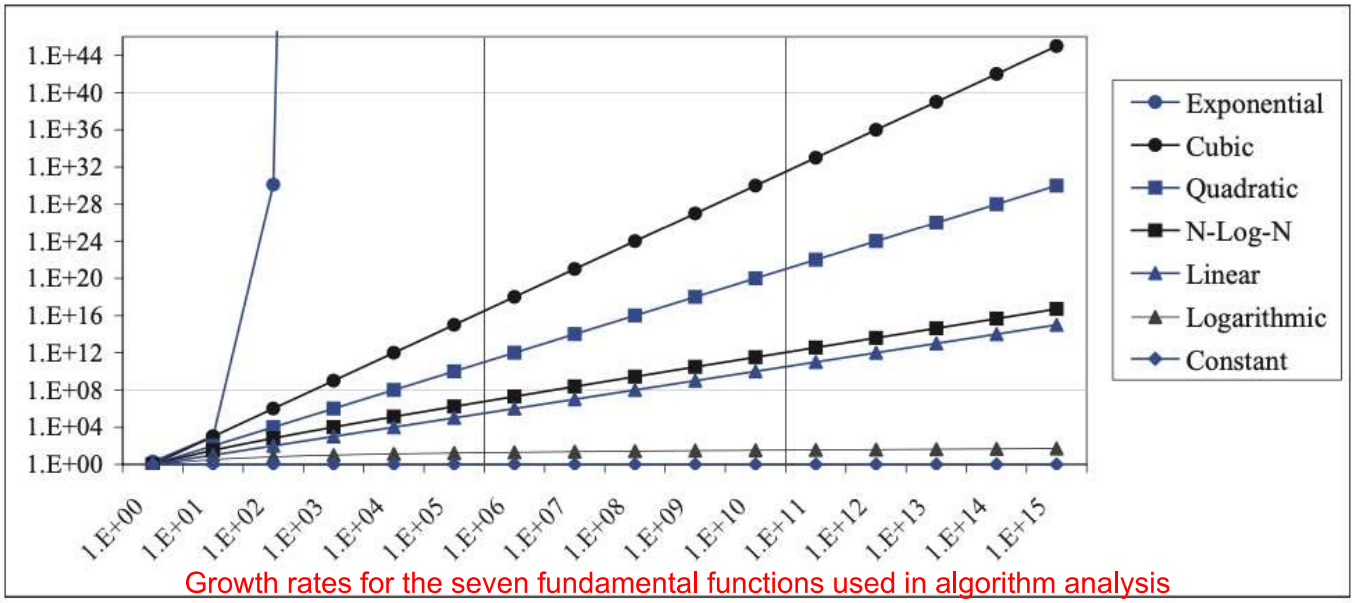
Why Growth Rate Matters

- **Classes of functions**

constant	*logarithm*	*linear*	*n-log-n*	*quadratic*	*cubic*	*exponential*
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

- Ideally, we would like:
  - data structure operations to run in times proportional to the **constant** or **logarithm** function.
  - algorithms to run in linear or n-log-n time.
  - Algorithms with quadratic or cubic running times are less practical, but algorithms with exponential running times are infeasible for all but the small-sized inputs.

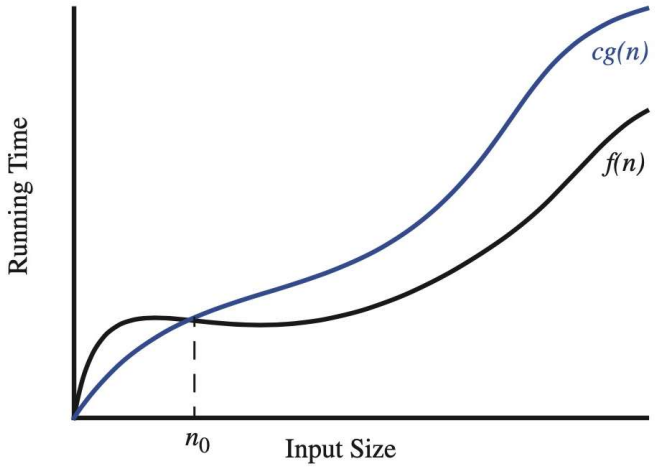
log-log chart

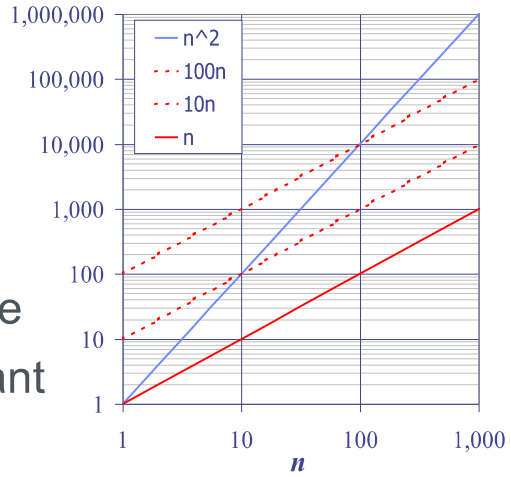


Growth rates for the seven fundamental functions used in algorithm analysis

Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that: $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
    - $(c - 2)\, n \geq 10$
    - $n \geq 10/(c - 2)$
    - Pick $c = 3$ and $n_0 = 10$
- Example: ArrayMax: $8n - 3$ is $O(n)$
  - $8n - 3 \leq cn$
  - Pick $c = 8$, and $n_0 = 1$
- Example: n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant





Asymptotic Analysis of Algorithms

- Now we can write the following mathematically precise statement on the running time of algorithm arrayMax for ***any*** computer:
  - The Algorithm arrayMax, for computing the maximum element in an array of **n** integers, runs in **O(n)** time**.**
  - proof: The number of primitive operations executed by algorithm **arrayMax** in each iteration is a constant. Hence, since each primitive operation runs in constant time, we can say that the running time of algorithm **arrayMax** on an input of size **n** is **at most a constant times n**, that is, we may conclude that the **running time of algorithm arrayMax is O(n).**
- The asymptotic analysis
  - identify the running time in Big-Oh notation
  - We find the worst-case number of primitive

  operations executed as a function of the input size
  - We express this function with Big-Oh notation

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.
$currMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

Big-Oh Notation Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$:
  - Drop lower-order terms
  - Drop constant factors
    - $3n^3 + 20n^2 + 5$ is $O(n^3)$
      - need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$ this is true for $c = 4$ and $n_0 = 21$
- Use the **smallest possible class** of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the **simplest expression of the class**
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"
- Think about hidden constant factors!

The big-Oh notation gives an upper bound on the growth rate of a function

Asymptotic Analysis - Example

- Prefix Averages
- The i-th prefix average of an array X is average of the first $(i+1)$ elements of X:
  $A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Algorithm prefixAverages1(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0],\ldots,X[i]$.

```
Let A be an array of n numbers.
for i ← 0 to n − 1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]
    A[i] ← a/(i + 1)
return array A
```

Algorithm prefixAverages2(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0],\ldots,X[i]$.

```
Let A be an array of n numbers.
s ← 0
for i ← 0 to n − 1 do
    s ← s + X[i]
    A[i] ← s/(i + 1)
return array A
```

Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

17 Algorithms Analysis (cont.2)

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

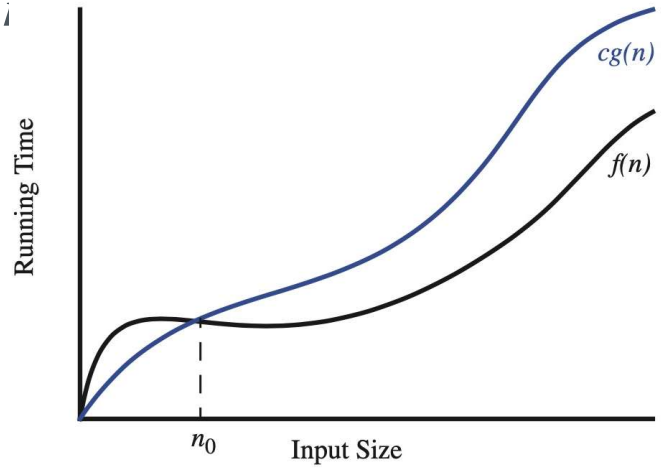
March 2, 2022

Administration

- My Office Hour:
  - Today at 15:00 in ITB-159 in-person (or virtually using teams)
- Please **read the questions carefully** and watch the video if needed for the assignment 2

Review

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that: $f(n) \leq cg(n)$ for
- Example: $2n + 10$ is $O(n)$
- Example: ArrayMax: $8n - 3$ is $O(n)$
- Example: n^2 is not $O(n)$



- Known class of functions:

constant	*logarithm*	*linear*	*n-log-n*	*quadratic*	*cubic*	*exponential*
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

- Desired complexities:
  - Linear or n-log-n for algorithms
    - sort, search
  - Constant or Logarithm operations for DS
    - add, remove, indexing

Asymptotic Analysis of Algorithms

- Now we can write the following mathematically precise statement on the running time of algorithm arrayMax for ***any*** computer:
  - The Algorithm arrayMax, for computing the maximum element in an array of **n** integers, runs in **O(n)** time**.**
  - proof: The number of primitive operations executed by algorithm **arrayMax** in each iteration is a constant. Hence, since each primitive operation runs in constant time, we can say that the running time of algorithm **arrayMax** on an input of size **n** is **at most a constant times n**, that is, we may conclude that the **running time of algorithm arrayMax is O(n).**
- The asymptotic analysis
  - identify the running time in Big-Oh notation
  - We find the worst-case number of primitive

  operations executed as a function of the input size
  - We express this function with Big-Oh notation

Algorithm arrayMax(A,n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.
$currMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

Big-Oh Notation Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$:
  - Drop lower-order terms
  - Drop constant factors
    - $3n^3 + 20n^2 + 5$ is $O(n^3)$
      - need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$ this is true for $c = 4$ and $n_0 = 21$

```
for (i = 0; i < n; i++) {
    sequence of statements
}
```

is O(n)

Big-Oh Notation Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$:
  - Drop lower-order terms
  - Drop constant factors
    - $3n^3 + 20n^2 + 5$ is $O(n^3)$
      - need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$ this is true for $c = 4$ and $n_0 = 21$

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sequence of statements
    }
}
```

is $O(n^2)$

Big-Oh Notation Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$:
  - Drop lower-order terms
  - Drop constant factors
    - $3n^3 + 20n^2 + 5$ is $O(n^3)$
      - need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$ this is true for $c = 4$ and $n_0 = 21$

```
for (i = 0; i < n; i++) {
    for (j = 0; j < i; j++) {
        sequence of statements
    }
}
```

- The outer loops is O(n)
- The statements in the inner loop executed i+1 times: i.e.: 1 + 2 + 3 + … + n times which is n(n+1)/2 which is $O(n^2)$

So, this is $O(n^2)$

Big-Oh Notation Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$:
  - Drop lower-order terms
  - Drop constant factors
    - $3n^3 + 20n^2 + 5$ is $O(n^3)$
      - need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$ this is true for $c = 4$ and $n_0 = 21$

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sequence of statements
    }
}
for (k = 0; k < n; k++) {
    sequence of statements
}
```

is $O(n^2+n)$ which is $O(n^2)$
we select the main component that affects the growth

Big-Oh Notation Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$:
  - Drop lower-order terms
  - Drop constant factors
    - $3n^3 + 20n^2 + 5$ is $O(n^3)$
      - need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$ this is true for $c = 4$ and $n_0 = 21$

```
for (j = 0; j < n; j++)
    f(j);
```

is O(n)

- function ***f*** takes a constant time

```
for (j = 0; j < n; j++)
    g(j);
```

is $O(n^2)$

- function ***g*** takes a linear time proportional to its parameter

```
for (j = 0; j < n; j++)
    g(k);
```

is O(k•n)
or O(n), if ***k*** is not very large or has a relative size to ***n***

Big-Oh Notation Rules

- Use the **smallest possible class** of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the **simplest expression of the class**
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"
- Think about hidden constant factors!

The big-Oh notation gives an upper bound on the growth rate of a function

Complex Examples on Growth Rates

- You need some math to reason about some functions
- math you need to review
  - properties of logarithms:
    - $\log_b(x\ y) = \log_b x + \log_b y$
    - $\log_b\ (x / y) = \log_b x - \log_b y$
    - $\log_b x^a = a \log_b x$
    - $\log_b a = \log_x a / \log_x b$
  - properties of exponentials:
    - $a^{(b+c)} = a^b\ a^c$
    - $a^{bc} = (a^b)^c$
    - $a^b / a^c = a^{(b-c)}$
    - $b = a^{log_a b}$
    - $b^c = a^{c log_a b}$

for example:

$$(\sqrt{2})^{logn} = (2^{0.5})^{logn} = (2^{logn})^{0.5} = \sqrt{n}$$

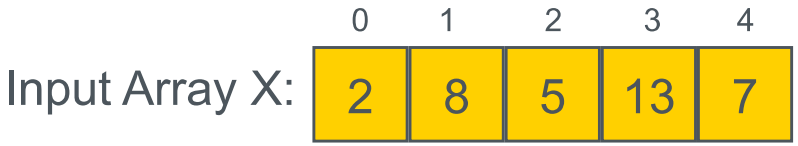
$$2^{100logn} = 2^{logn^{100}} = n^{100}$$

$$\log(n!) = \log(n(n-1)(n-2)\ldots) = \log(n) + \log(n-1) + \cdots = \text{is } O(\log(n))$$

Asymptotic Analysis - Example

- Prefix Averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:
  $A[i] = (X[0] + X[1] + \ldots + X[i]) / (i+1)$

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$



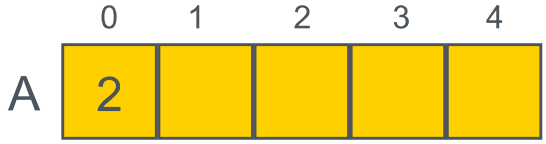
Asymptotic Analysis - Example

- Prefix Averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:
  $A[i] = (X[0] + X[1] + \ldots + X[i]) / (i+1)$

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Algorithm prefixAverages1(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0], \ldots, X[i]$.

```
Let A be an array of n numbers.
for i ← 0 to n − 1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]
    A[i] ← a/(i+1)
return array A
```



i = 0
a = 0
after inner for loop (i + 1 times)
a = 2
A[0] = 2 / 1 = 2

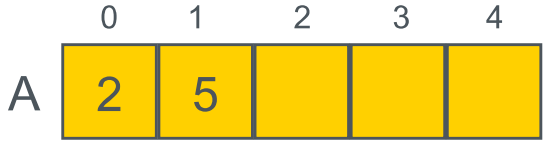
Asymptotic Analysis - Example

- Prefix Averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:
  $A[i] = (X[0] + X[1] + \ldots + X[i]) / (i+1)$

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Algorithm prefixAverages1(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0], \ldots, X[i]$.

```
Let A be an array of n numbers.
for i ← 0 to n − 1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]
    A[i] ← a/(i+1)
return array A
```



i = 1
a = 0
after inner for loop (i + 1 times)
a = 10
A[0] = 10 / 2 = 5

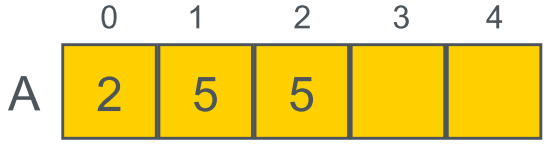
Asymptotic Analysis - Example

- Prefix Averages
- The i-th prefix average of an array X is average of the first $(i+1)$ elements of X:
  $A[i] = (X[0] + X[1] + \ldots + X[i]) / (i+1)$

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Algorithm prefixAverages1(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0], \ldots, X[i]$.

```
Let A be an array of n numbers.
for i ← 0 to n − 1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]
    A[i] ← a/(i+1)
return array A
```



i = 2
a = 0
after inner for loop (i + 1 times)
a = 15
A[0] = 15 / 3 = 5

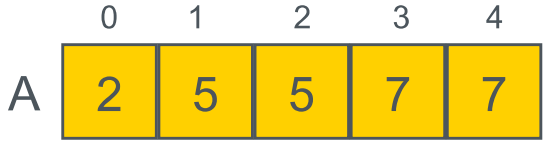
Asymptotic Analysis - Example

- Prefix Averages
- The i-th prefix average of an array X is average of the first $(i+1)$ elements of X:
  $A[i] = (X[0] + X[1] + \ldots + X[i]) / (i+1)$

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Algorithm prefixAverages1(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0], \ldots, X[i]$.

```
Let A be an array of n numbers.
for i ← 0 to n − 1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]
    A[i] ← a/(i+1)
return array A
```



Asymptotic Analysis - Example

- Prefix Averages
- The i-th prefix average of an array X is average of the first $(i+1)$ elements of X:
  $A[i] = (X[0] + X[1] + \ldots + X[i]) / (i+1)$

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Algorithm prefixAverages1(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0], \ldots, X[i]$.

```
Let A be an array of n numbers.  <---------- O(n)
for i ← 0 to n − 1 do  <----------O(n) operations
  a ← 0  <----------O(n) operations
  for j ← 0 to i do  <-- O(n^2) operations
    a ← a + X[j]  <-- O(n^2) operations
  A[i] ← a/(i+1)  <---------- O(n) operations
return array A  <-- O(1) operation (in the textbook it is O(n) considering return by value)
```

so prefixAverages1 is $O(n^2)$

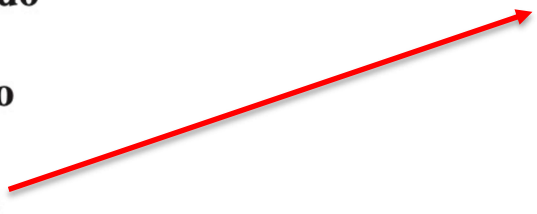
Asymptotic Analysis - Example

- Prefix Averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:
  $A[i] = (X[0] + X[1] + \ldots + X[i]) / (i+1)$

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Algorithm prefixAverages1(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0], \ldots, X[i]$.

Let A be an array of n numbers.
for $i \leftarrow 0$ **to** $n-1$ **do**
  $a \leftarrow 0$
  for $j \leftarrow 0$ **to** i **do**
    $a \leftarrow a + X[j]$
  $A[i] \leftarrow a/(i+1)$
return array A



A[i−1] = (X[0]+X[1]+⋯+X[i−1]) / i
A[i] = (X[0]+X[1]+⋯+X[i−1]+X[i]) / (i+1)

Asymptotic Analysis - Example

- Prefix Averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:
  $A[i] = (X[0] + X[1] + \ldots + X[i]) / (i+1)$

A[i−1] = (X[0]+X[1]+⋯+X[i−1]) / i
A[i] = (X[0]+X[1]+⋯+X[i−1]+X[i]) / (i+1)

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Algorithm prefixAverages2(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0], \ldots, X[i]$.

Let A be an array of n numbers. <---------- O(n)
$s \leftarrow 0$ <----------O(1) operations
for $i \leftarrow 0$ **to** $n-1$ **do** <--- O(n) operations
  $s \leftarrow s + X[i]$ <------- O(n) operations
  $A[i] \leftarrow s/(i+1)$ <----- O(n) operations
return array A <-- O(1) operation (in the textbook it is O(n) considering return by value)

so prefixAverages2 is O(n)

Using Sorting as a Problem-Solving Tool

- Sometimes we can use sorting as a tool to ease our problem-solving and even make the algorithm faster
- Element uniqueness problem:
  - Given an array, identify if all elements are unique (there are no duplicates)
    - Naïve algorithm:
      - compare each pair of elements in the array
        - $O(n^2)$
    - Better algorithm
      - sort the array first, then traverse the array once, look for duplicates among consecutive elements.
        - Best sorting algorithm runs in O(nlogn)
        - The algorithm will be O(nlogn + n) which is (nlogn)

Final Remarks

- Given an algorithm
  - What is the order of this algorithm? (runtime complexity)
- Try to reduce the running time as much as possible
  - Sometimes based on some observations that we already had we can easily reduce the complexity (like previous slide)
- This analysis technique is not the whole story!
  - There are situations where a worst-case $O(n^2)$ is faster than O(nlogn)
  - If you want, you can study more.
- There are some problems for which there is **NO** polynomial-time algorithm found (up until now)
  - We say that they “NP-hard” or “NP-complete”
  - If someone can find a polynomial-time solution one of them, that solution may work for many others as well.

Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

18 Stacks

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

March 3, 2022

Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

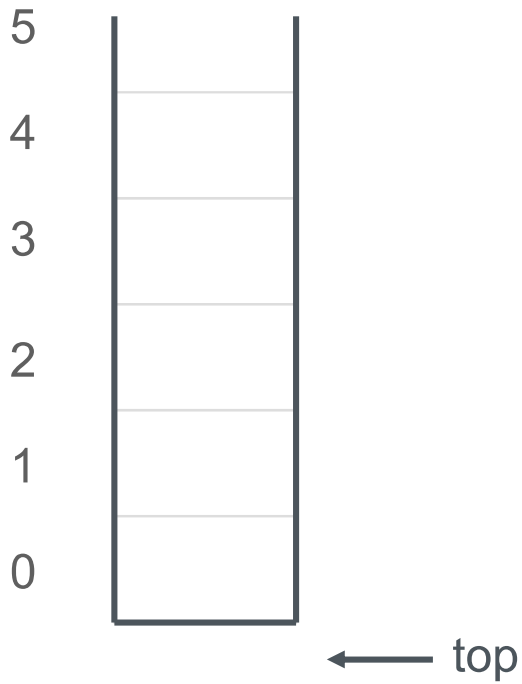
push (B)

push (C)

push (D)

push (E)

pop ()

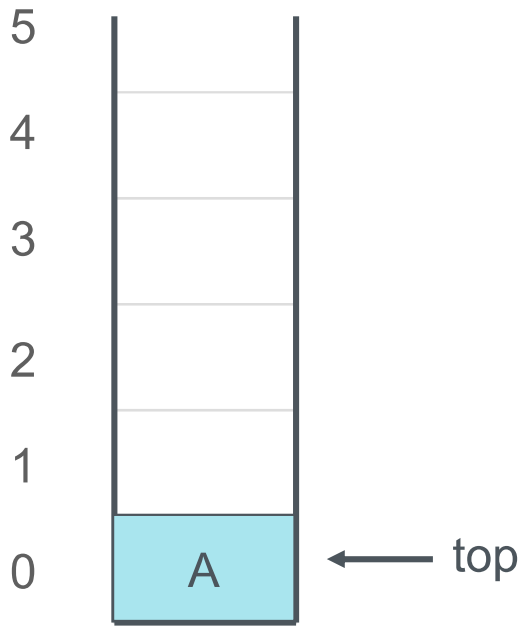


Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

- push (A)
- push (B)
- push (C)
- push (D)
- push (E)
- pop ()



Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

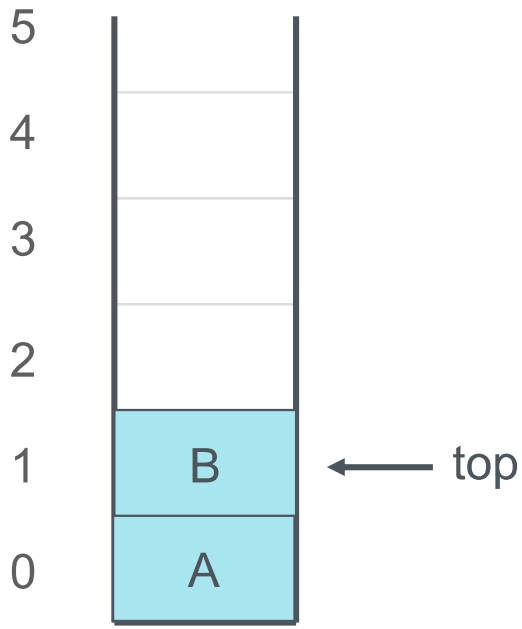
push (B)

push (C)

push (D)

push (E)

pop ()



Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

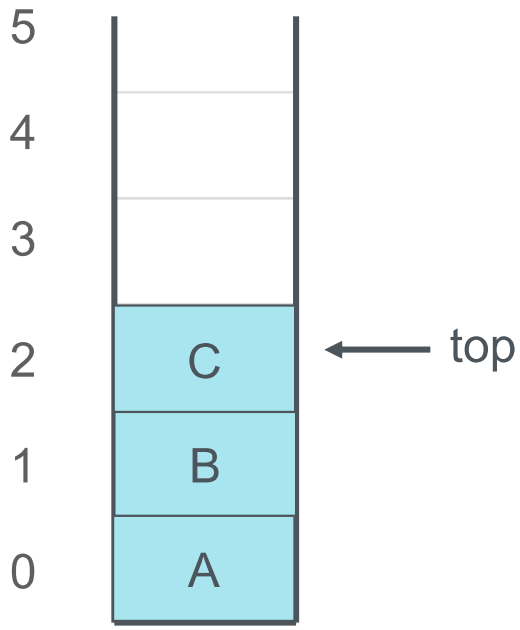
push (B)

push (C)

push (D)

push (E)

pop ()



Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

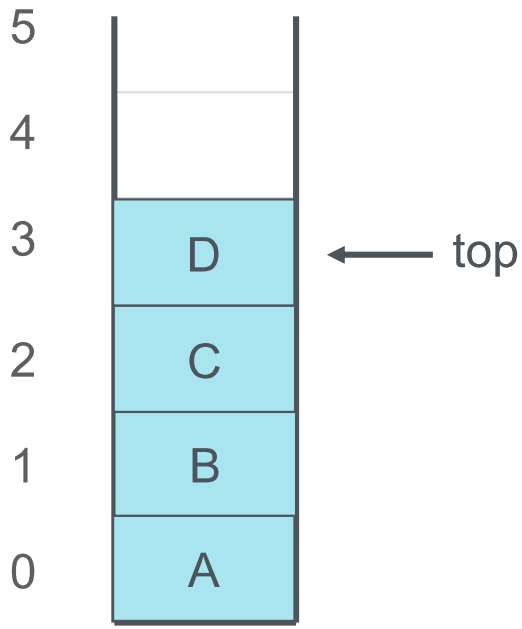
push (B)

push (C)

push (D)

push (E)

pop ()



Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

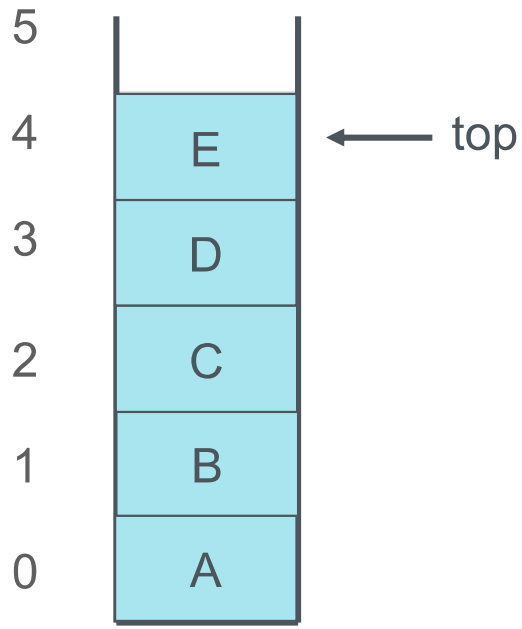
push (B)

push (C)

push (D)

push (E)

pop ()



Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

push (B)

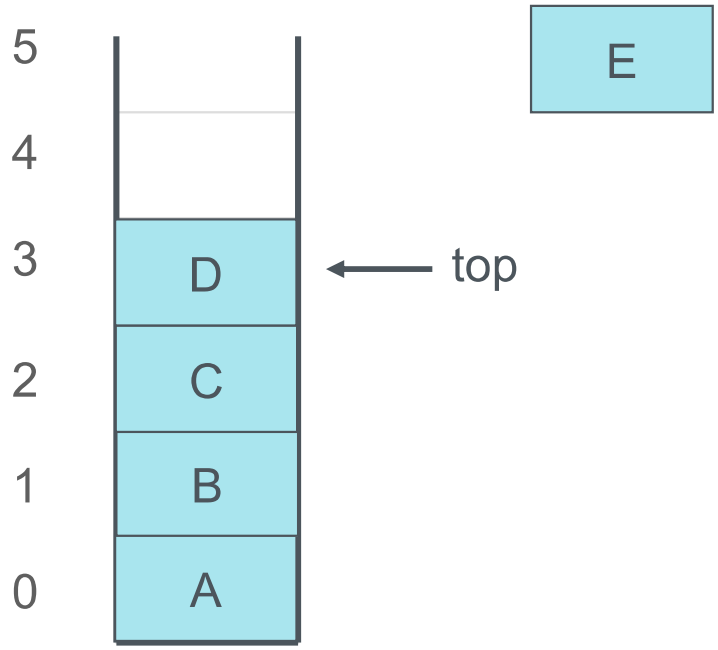
push (C)

push (D)

push (E)

- pop ()

Remove an item from the top of stack.



Stack and its Applications

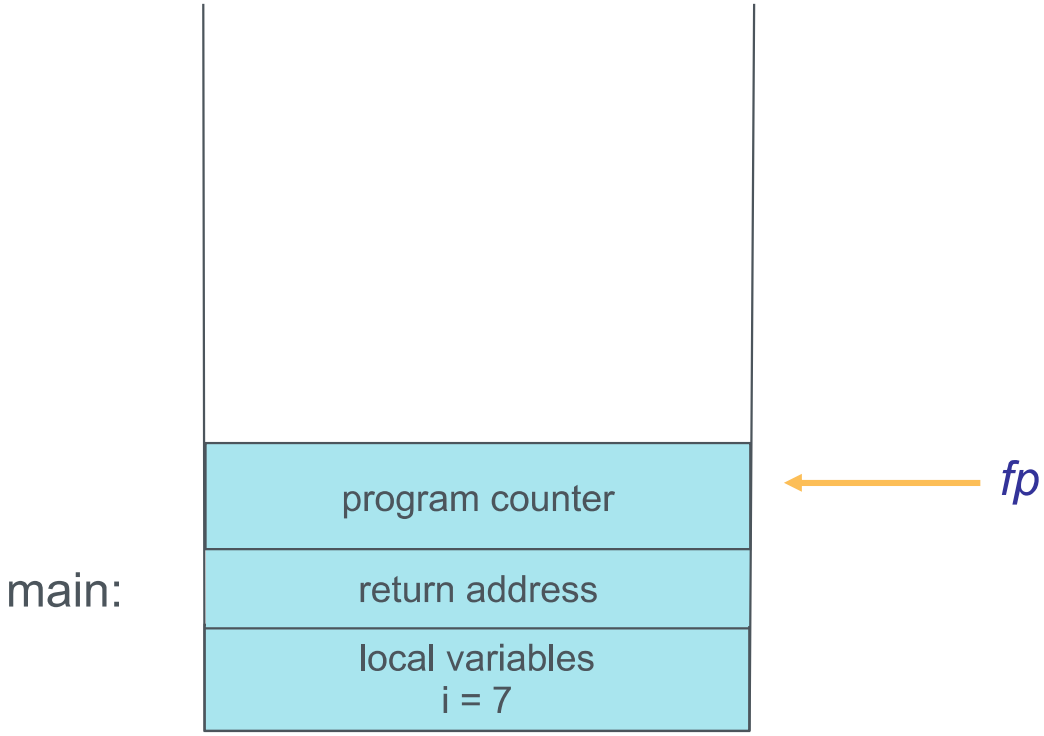
- Applications
  - History of visited pages in a web browser
  - Sequence of Undo operations in a text editor
  - Keep track of function calls in the C++ run-time system (System Stack)
  - As a data structure for algorithms to solve problems
    - Example: Reversing a list
  - Component of other data structures

Stack and its Applications

- System Stack
  - The C++ run-time system keeps track of the chain of active function calls with a stack.
- When a function is called, the system pushes a "**stack frame**" onto the stack which contains:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When the function ends, its frame is popped from the stack and control is passed to the function on top of the stack
- Allows for recursion

Stack and its Applications

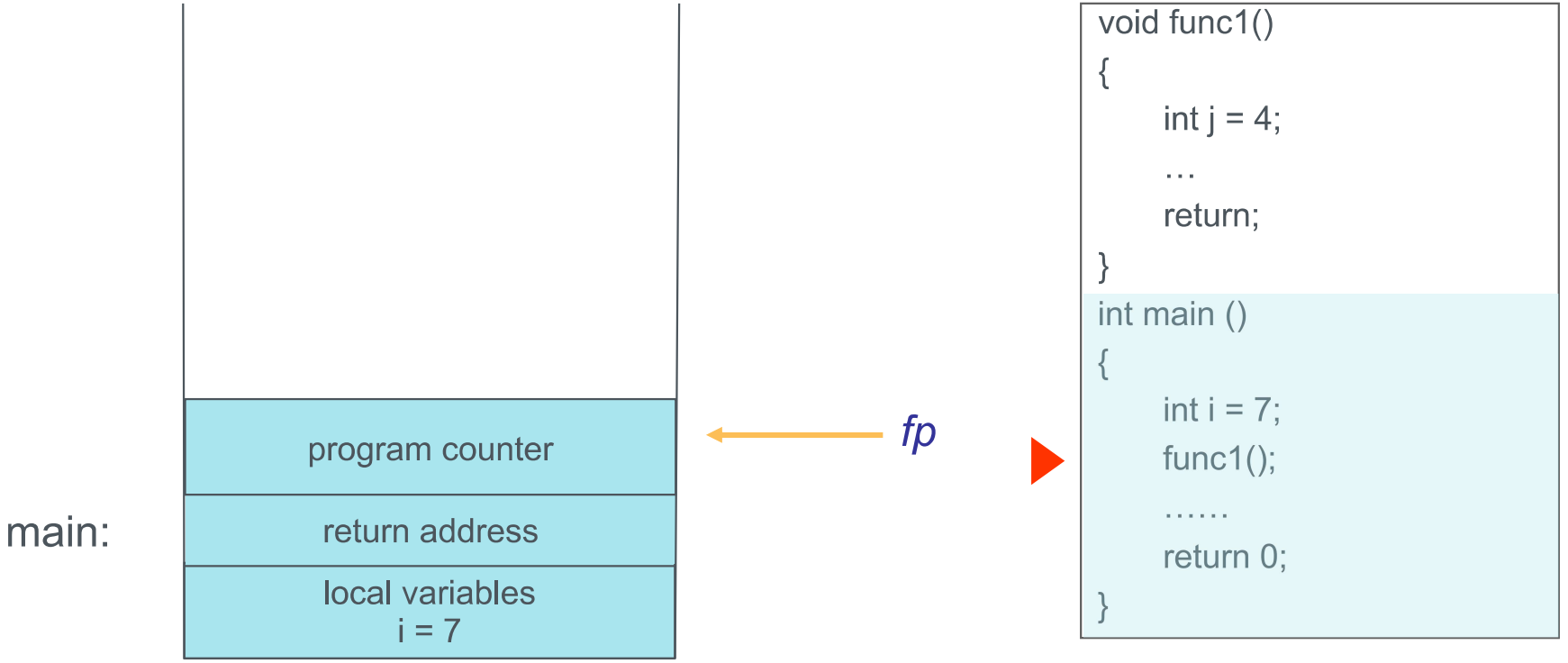
- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed




```
void func1()
{
      int j = 4;
      …
      return;
}
int main ()
{
      int i = 7;
      func1();
      ……
      return 0;
}
```

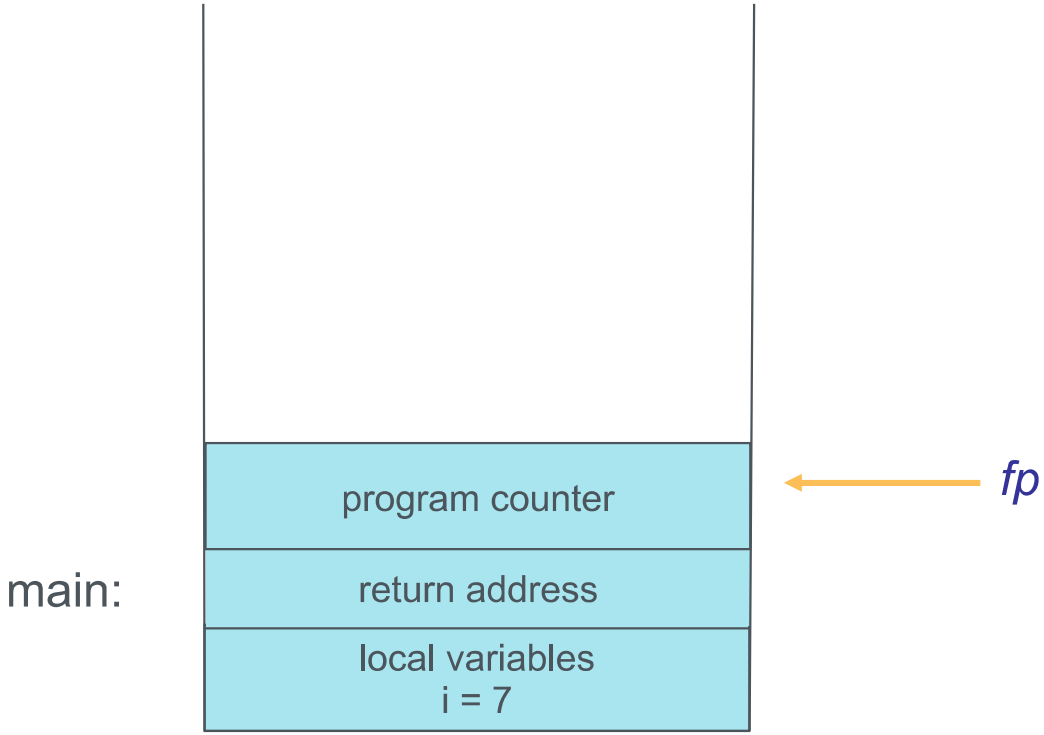
Stack and its Applications

- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed



Stack and its Applications

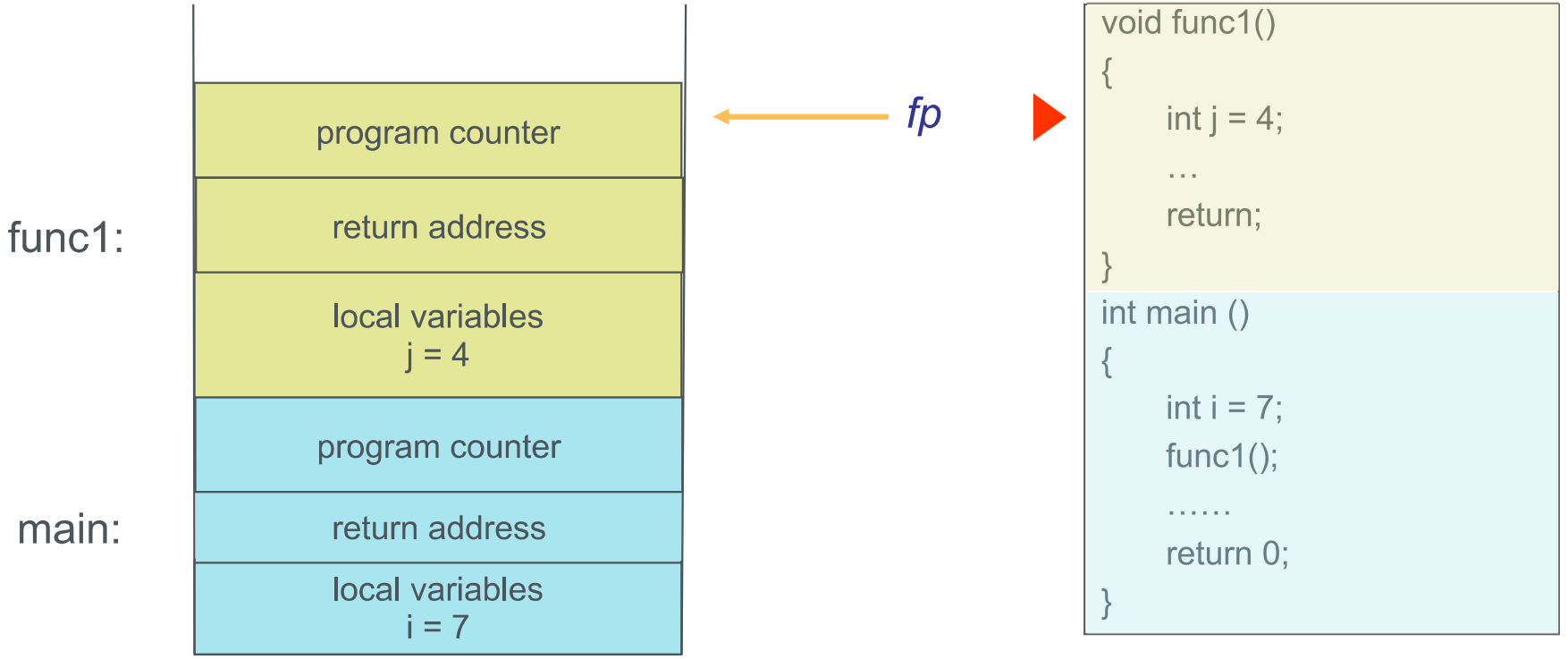
- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed




```
void func1()
{
      int j = 4;
      …
      return;
}
int main ()
{
      int i = 7;
      func1();
      ……
      return 0;
}
```

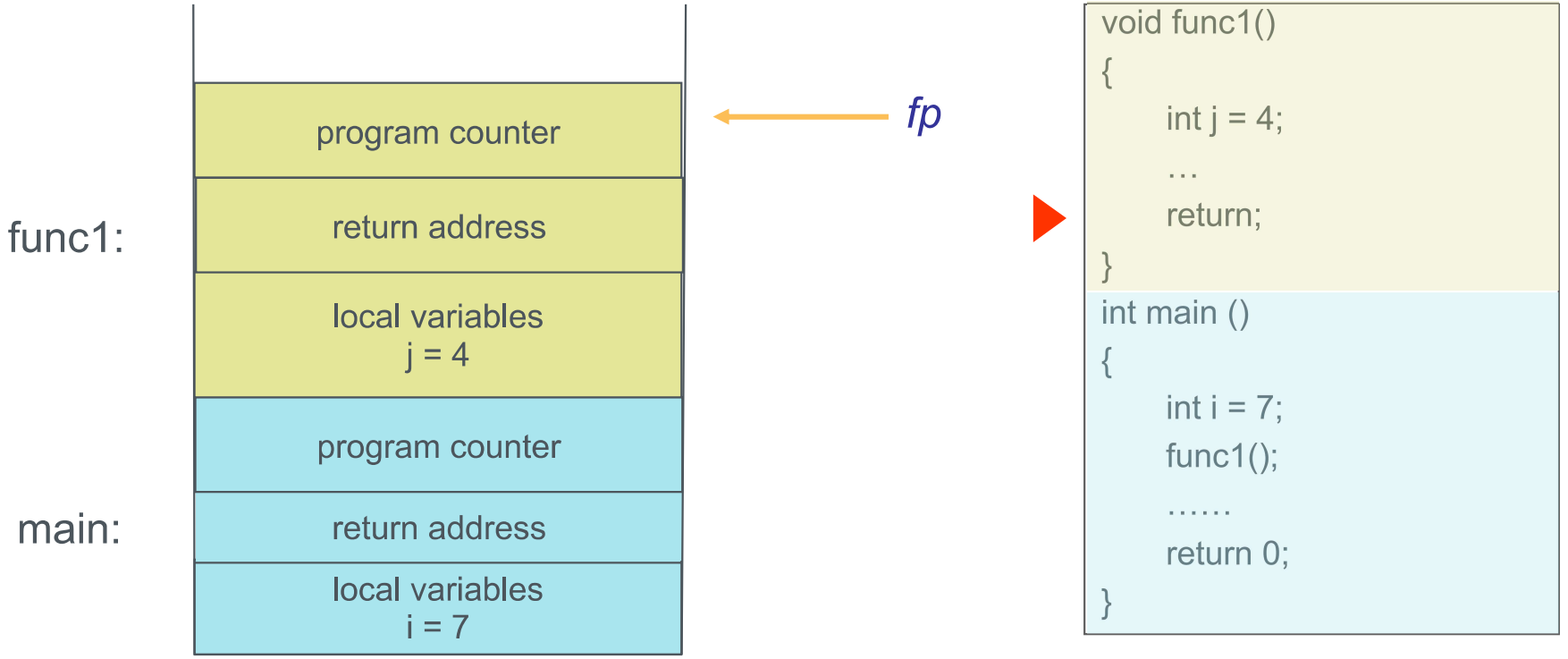
Stack and its Applications

- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed



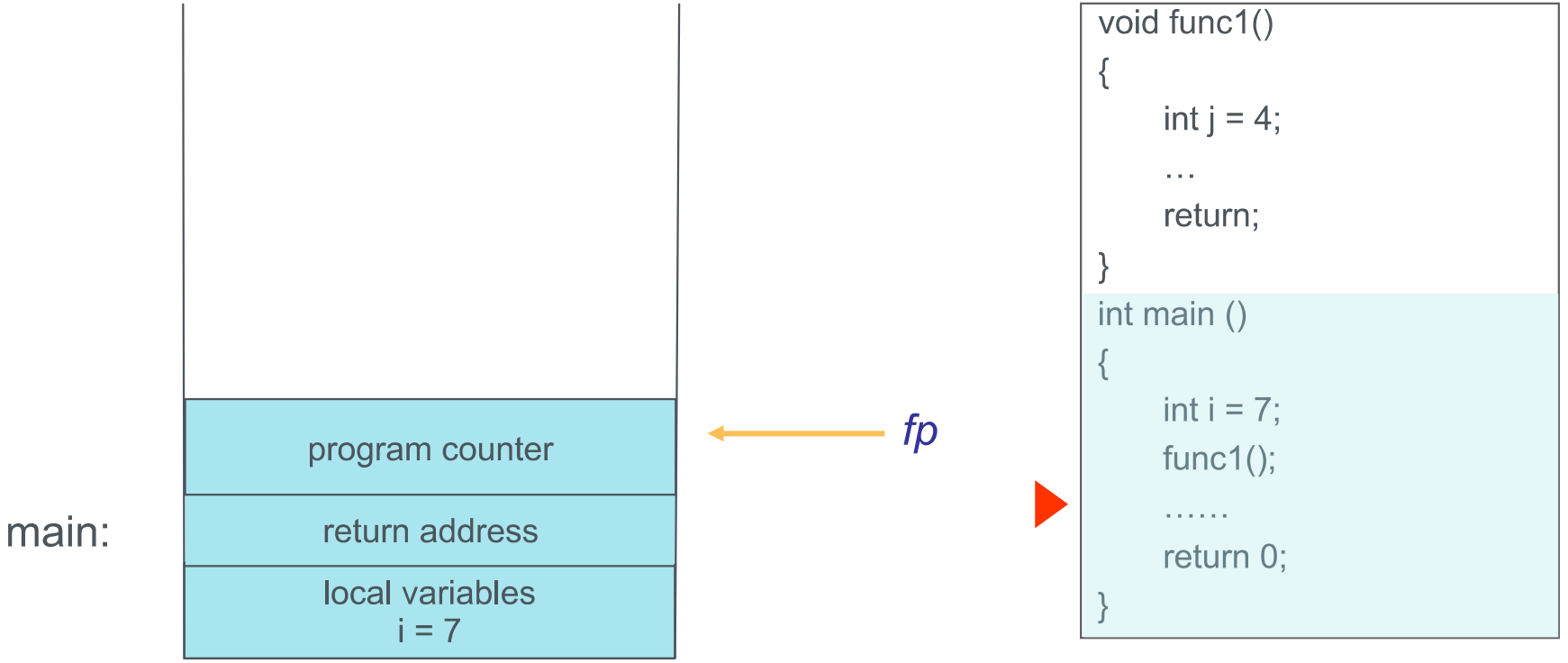
Stack and its Applications

- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed



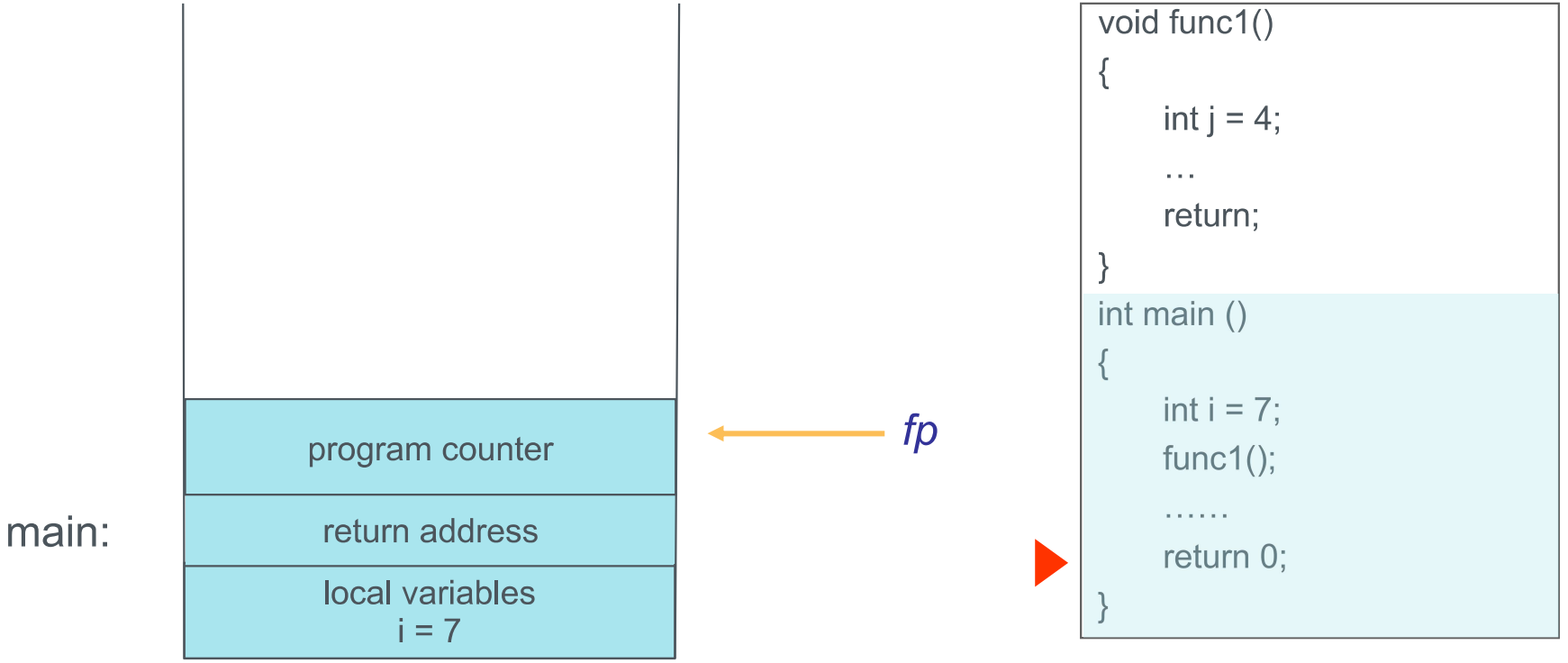
Stack and its Applications

- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed



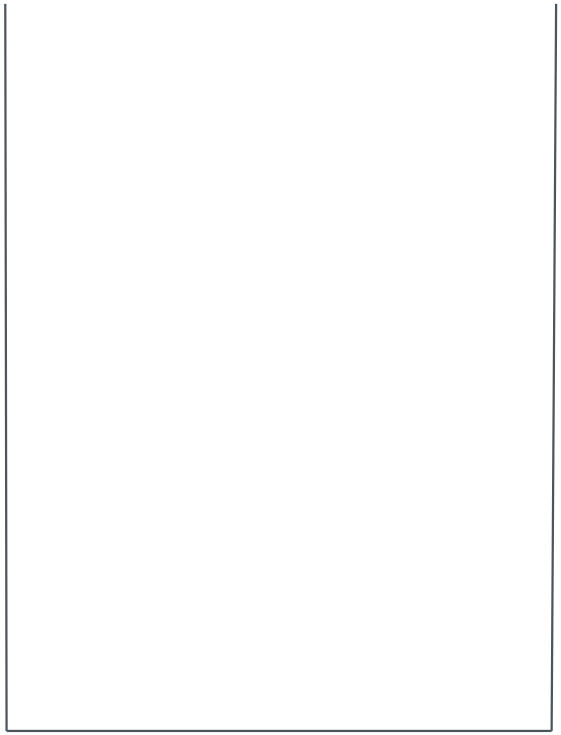
Stack and its Applications

- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed



Stack and its Applications

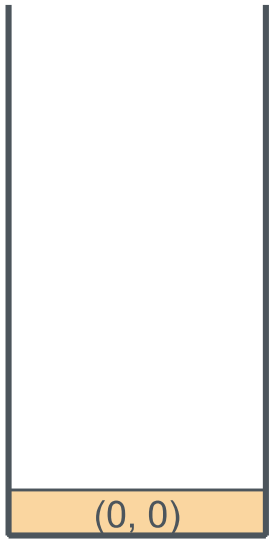
- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed



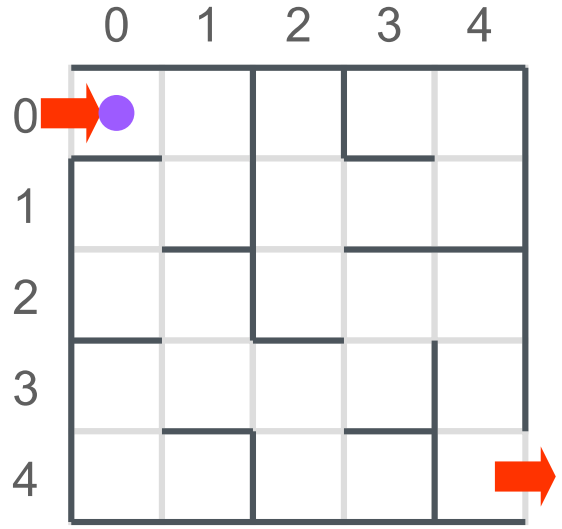

```
void func1()
{
        int j = 4;
        …
        return;
}
int main ()
{
        int i = 7;
        func1();
        ……
        return 0;
}
```

Stack and its Applications

- Path Planning for a Robot

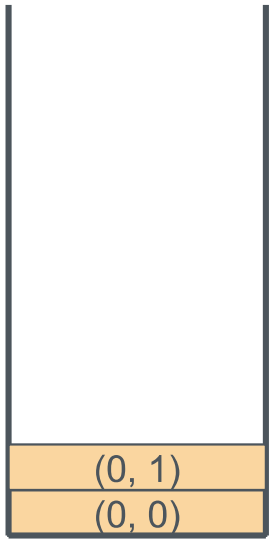


Stack

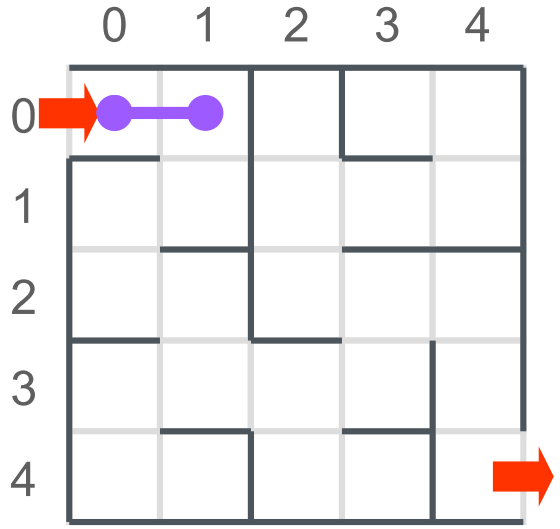


Stack and its Applications

- Path Planning for a Robot



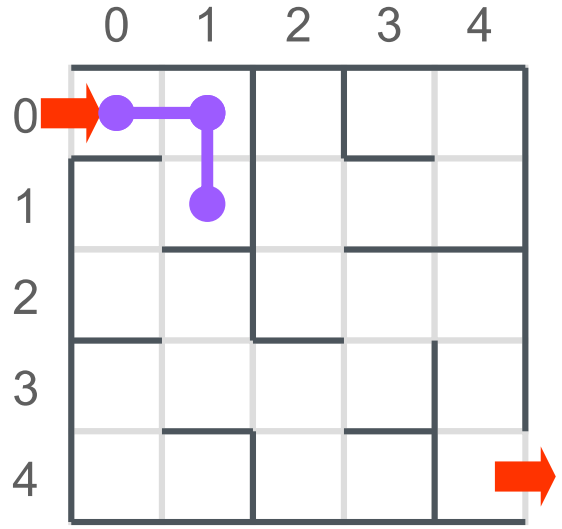
Stack



Stack and its Applications

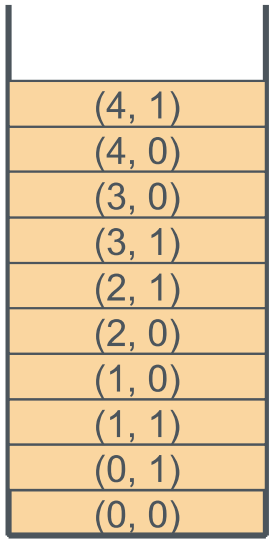
- Path Planning for a Robot

Stack

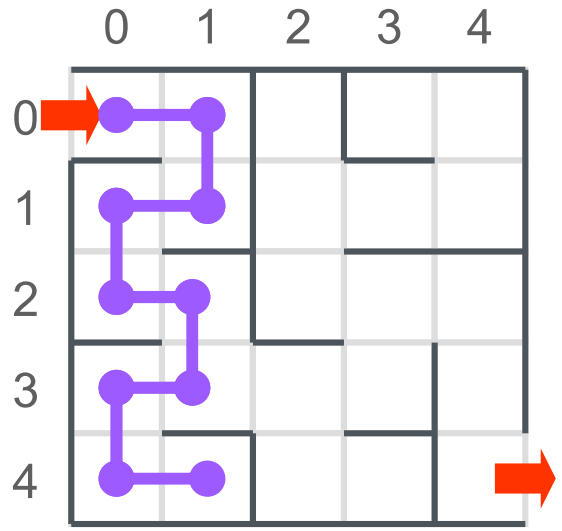


Stack and its Applications

- Path Planning for a Robot

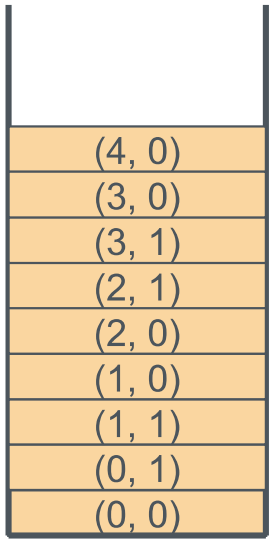


Stack

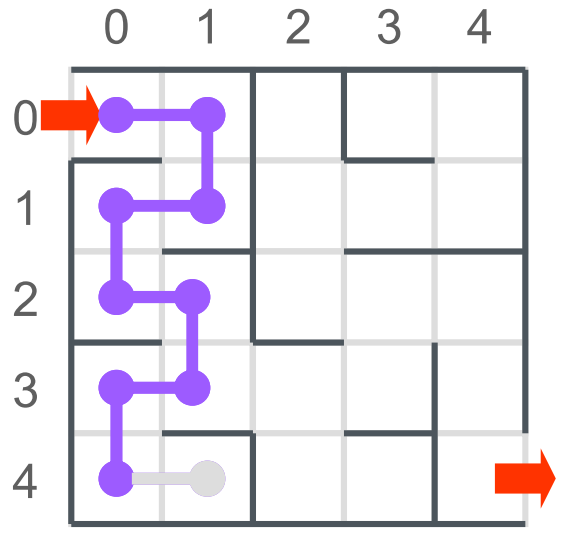


Stack and its Applications

- Path Planning for a Robot

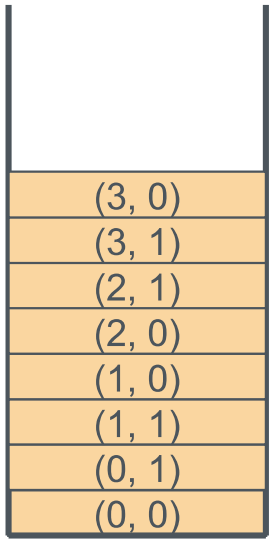


Stack

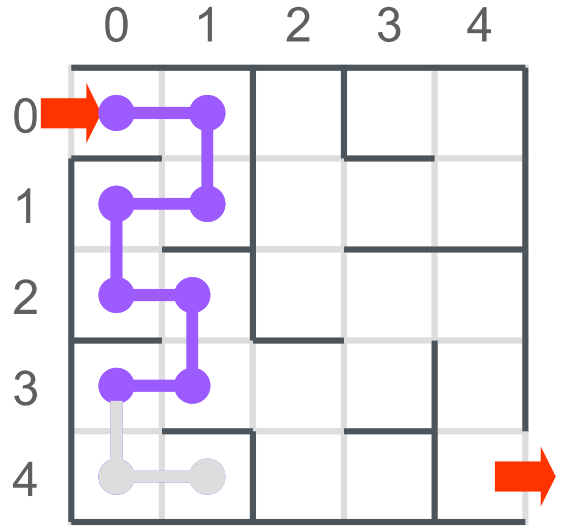


Stack and its Applications

- Path Planning for a Robot

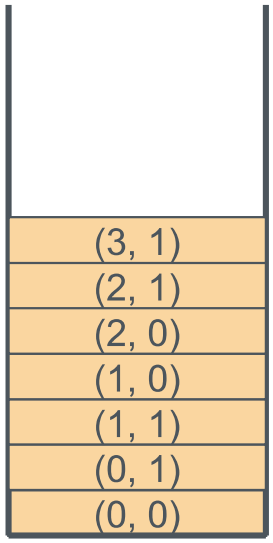


Stack

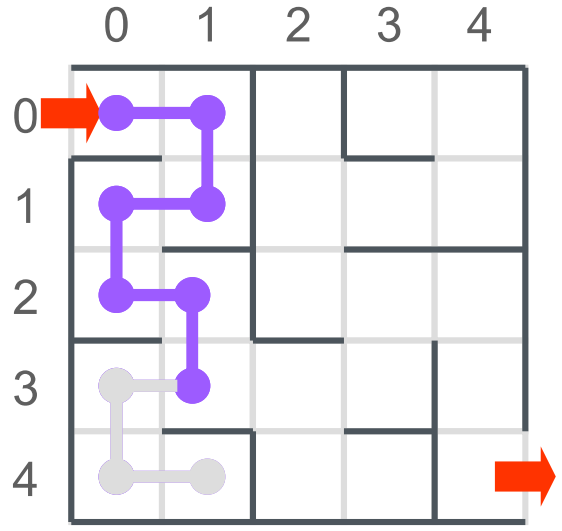


Stack and its Applications

- Path Planning for a Robot

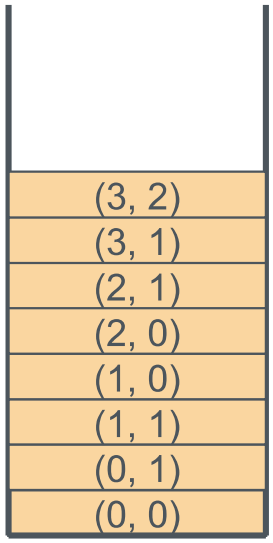


Stack

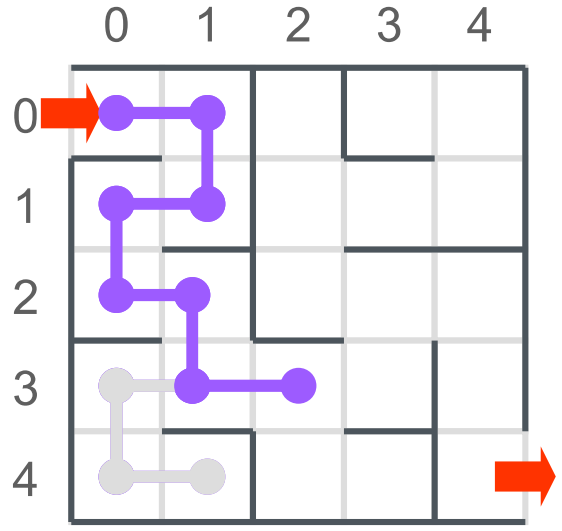


Stack and its Applications

- Path Planning for a Robot



Stack



ADT

- An ADT is a mathematical model (abstraction) of a data structure that specifies
  - the type of the data stored
  - the operations supported on them
  - the types of the parameters of the operations.
  - Error conditions associated with operations
- An **ADT specifies what each operation does**, but **not how** it does it.

The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Main stack operations:
  - push(object): inserts an element object
  - pop(): removes the top element
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored in stack
  - Boolean empty(): indicates whether no elements are stored

```
template <typename E>
class Stack {                                   // an interface for a stack
public:
  int size() const;                             // number of items in stack
  bool empty() const;                           // is the stack empty?
  const E& top() const throw(StackEmpty);       // the top element
  void push(const E& e);                        // push x onto the stack
  void pop() throw(StackEmpty);                 // remove the top element
};
```

Exceptions for an ADT

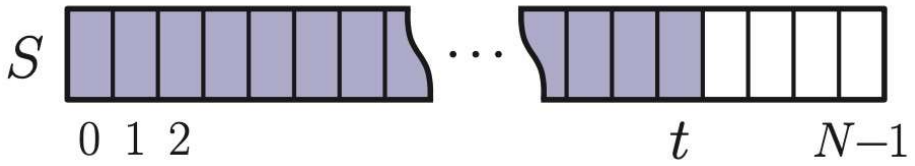
- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "**thrown**" by an operation that cannot be executed

- In the Stack ADT, operations **pop** and **top** cannot be performed if the stack is empty
- Attempting **pop** or **top** on an empty stack throws a **StackEmpty** exception

```
// Exception thrown on performing top or pop of an empty stack.
class StackEmpty : public RuntimeException {
public:
  StackEmpty(const string& err) : RuntimeException(err) {}
};
```

Array-Based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element



- The array storing the stack elements may become **full**
- A push operation will then throw a **StackFull** exception:
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm size():
    return $t + 1$
Algorithm empty():
    return $(t < 0)$
Algorithm top():
    if empty() **then**
        throw StackEmpty exception
    return $S[t]$
Algorithm push(e):
    if size() $= N$ **then**
        throw StackFull exception
    $t \leftarrow t + 1$
    $S[t] \leftarrow e$
Algorithm pop():
    if empty() **then**
        throw StackEmpty exception
    $t \leftarrow t - 1$

Array-Based Stack - C++ Implementation

```
template <typename E>
class ArrayStack {
  enum { DEF_CAPACITY = 100 };          // default stack capacity
public:
  ArrayStack(int cap = DEF_CAPACITY);   // constructor from capacity
  int size() const;                     // number of items in the stack
  bool empty() const;                   // is the stack empty?
  const E& top() const throw(StackEmpty); // get the top element
  void push(const E& e) throw(StackFull); // push element onto stack
  void pop() throw(StackEmpty);         // pop the stack
  // ...housekeeping functions omitted
private:                                // member data
  E* S;                                 // array of stack elements
  int capacity;                         // stack capacity
  int t;                                // index of the top of the stack
};
```

Array-Based Stack - C++ Implementation

```
template <typename E> ArrayStack<E>::ArrayStack(int cap)
  : S(new E[cap]), capacity(cap), t(-1) { } // constructor from capacity

template <typename E> int ArrayStack<E>::size() const
  { return (t + 1); }                      // number of items in the stack

template <typename E> bool ArrayStack<E>::empty() const
  { return (t < 0); }                      // is the stack empty?

template <typename E>                      // return top of stack
const E& ArrayStack<E>::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S[t];
}

template <typename E>                      // push element onto the stack
void ArrayStack<E>::push(const E& e) throw(StackFull) {
  if (size() == capacity) throw StackFull("Push to full stack");
  S[++t] = e;
}

template <typename E>                      // pop the stack
void ArrayStack<E>::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  --t;
}
```

Array-Based Stack - C++ Implementation

- Example use:

```
ArrayStack<int> A;                          // A = [], size = 0
A.push(7);                                  // A = [7*], size = 1
A.push(13);                                 // A = [7, 13*], size = 2
cout << A.top() << endl; A.pop();           // A = [7*], outputs: 13
A.push(9);                                  // A = [7, 9*], size = 2
cout << A.top() << endl;                    // A = [7, 9*], outputs: 9
cout << A.top() << endl; A.pop();           // A = [7*], outputs: 9
ArrayStack<string> B(10);                   // B = [], size = 0
B.push("Bob");                              // B = [Bob*], size = 1
B.push("Alice");                            // B = [Bob, Alice*], size = 2
cout << B.top() << endl; B.pop();           // B = [Bob*], outputs: Alice
B.push("Eve");                              // B = [Bob, Eve*], size = 2
```

Array-Based Stack - Performance

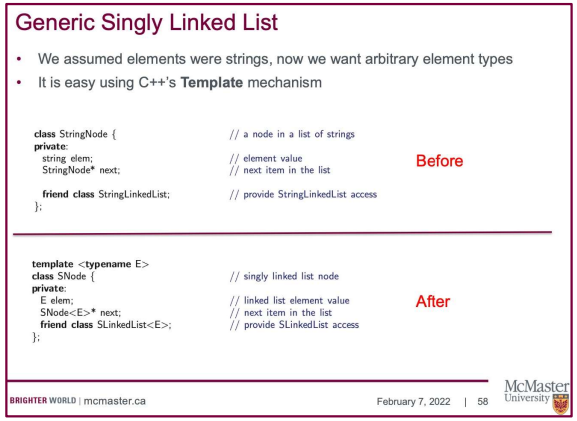
- Performance:
  - Let **n** be the number of elements in the stack
  - The space used is **O(n)**
  - Each operation runs in time **O(1)**

Operation	*Time*
size	$O(1)$
empty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

- Limitations:
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Generic Linked List-Based Implementation

- We have already developed:
  - Generic Singly Linked List (Lecture of Feb 7)
- We use that SLinkedList class
  - **S** stores the stack values
  - **n** stores the number of elements on stack




```
typedef string Elem;                          // stack element type
class LinkedStack {                           // stack as a linked list
public:
  LinkedStack();                              // constructor
  int size() const;                           // number of items in the stack
  bool empty() const;                         // is the stack empty?
  const Elem& top() const throw(StackEmpty); // the top element
  void push(const Elem& e);                   // push element onto stack
  void pop() throw(StackEmpty);               // pop the stack
private:                                      // member data
  SLinkedList<Elem> S;                        // linked list of elements
  int n;                                      // number of elements
};
```

Generic Linked List-Based Implementation

```
LinkedStack::LinkedStack()
  : S(), n(0) { }                          // constructor

int LinkedStack::size() const
  { return n; }                            // number of items in the stack

bool LinkedStack::empty() const
  { return n == 0; }                       // is the stack empty?

                                           // get the top element
const Elem& LinkedStack::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S.front();
}
void LinkedStack::push(const Elem& e) {    // push element onto stack
  ++n;
  S.addFront(e);
}
                                           // pop the stack
void LinkedStack::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  --n;
  S.removeFront();
}
```

Parentheses Matching Problem

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Problem

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Algorithm ParenMatch(X, n):

Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: **true** if and only if all the grouping symbols in X match

```
Let S be an empty stack
for i ← 0 to n − 1 do
    if X[i] is an opening grouping symbol then
        S.push(X[i])
    else if X[i] is a closing grouping symbol then
        if S.empty() then
            return false          {nothing to match with}
        if S.top() does not match the type of X[i] then
            return false          {wrong type}
        S.pop()
if S.empty() then
    return true                   {every symbol matched}
else
    return false                  {some symbols were never matched}
```

Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

19 Queues

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

March 7, 2022

Stack Example - Parentheses Matching Problem

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Stack Example - Parentheses Matching Problem

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Algorithm ParenMatch(X, n):

Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

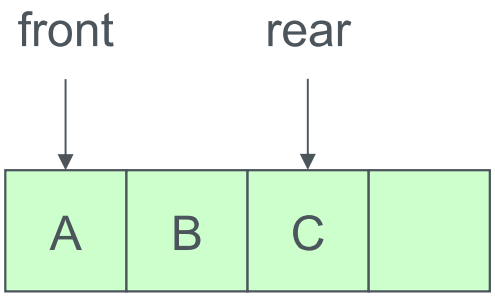
Output: **true** if and only if all the grouping symbols in X match

```
Let S be an empty stack
for i ← 0 to n − 1 do
    if X[i] is an opening grouping symbol then
        S.push(X[i])
    else if X[i] is a closing grouping symbol then
        if S.empty() then
            return false          {nothing to match with}
        if S.top() does not match the type of X[i] then
            return false          {wrong type}
        S.pop()
if S.empty() then
    return true                   {every symbol matched}
else
    return false                  {some symbols were never matched}
```

Queue

- A queue is a container of elements that are inserted and removed according to the First-In First-Out (FIFO) principle.
- Elements enter queue at the **rear** and *are removed* from the **front**

  "push" adds a new item on top of the stack.



Queue and its Applications

- Applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
  - As a data structure for algorithms to solve problems
    - BFS(Breadth First Search) for graphs
  - Component of other data structures

The Queue ADT

- The Queue ADT stores arbitrary objects
- Main stack operations:
  - enqueue(e): Insert element **e** at the rear of the queue
  - dequeue(): Remove element at the front of the queue; an error occurs if the queue is empty.
  - front(): Return, **but do not remove**, a reference to the front element in the queue; an error occurs if the queue is empty.
  - size(): Return the number of elements in the queue
  - empty(): Return **true** if the queue is empty and **false** otherwise.

```
template <typename E>
class Queue {                                  // an interface for a queue
public:
  int size() const;                            // number of items in queue
  bool empty() const;                          // is the queue empty?
  const E& front() const throw(QueueEmpty);    // the front element
  void enqueue (const E& e);                   // enqueue element at rear
  void dequeue() throw(QueueEmpty);            // dequeue element at front
};
```

Exceptions for Queue ADT

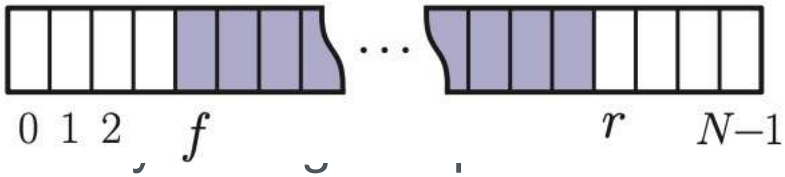
- In the Queue ADT, operations **dequeue** and **front** cannot be performed if the Queue is empty
- Attempting **dequeue** or **front** on an empty queue throws a **QueueEmpty** exception

```
class QueueEmpty : public RuntimeException {
public:
  QueueEmpty(const string& err) : RuntimeException(err) { }
};
```

```
class RuntimeException {                    // generic run-time exception
private:
  string errorMsg;
public:
  RuntimeException(const string& err) { errorMsg = err; }
  string getMessage() const { return errorMsg; }
};
```

Array-Based Queue

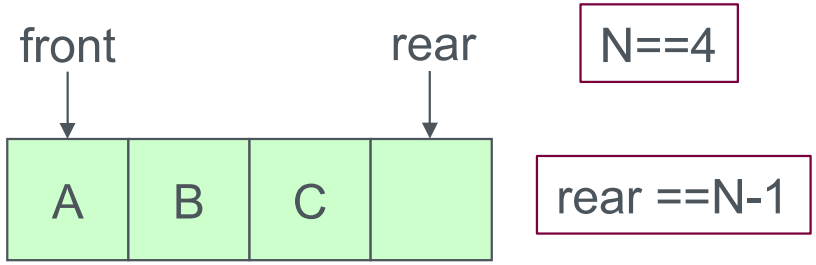
- An array with capacity N stores elements
- Two variables keeps track of the index of the front and rear element;
- **f**: index of front element
- **r**: index of element following rear element



- Th s may become **full**
- A **enqueue** operation will then throw a **QueueFull** exception

Array-Based Queue - An Issue

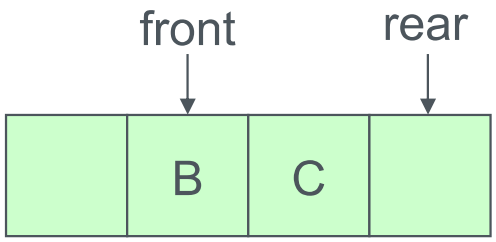
- **f**: index of front element
- **r**: index of element following rear element
- We can run into problem



N==4

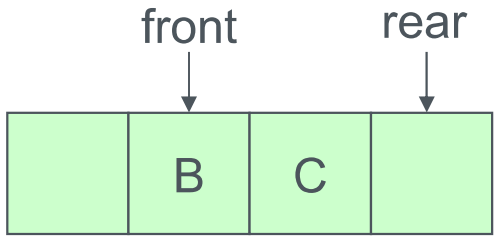
rear ==N-1

after dequeue



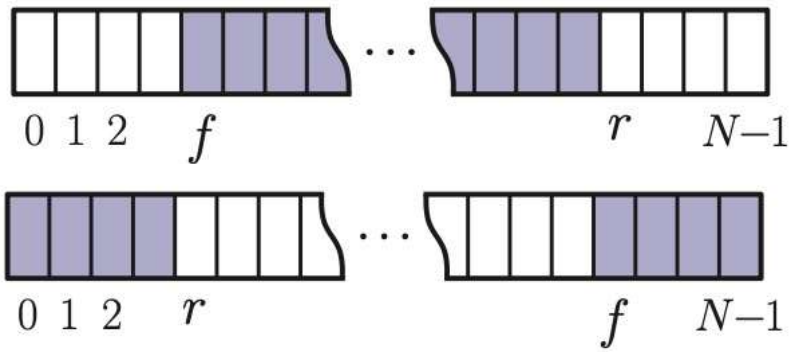
rear ==N-1

Space available!



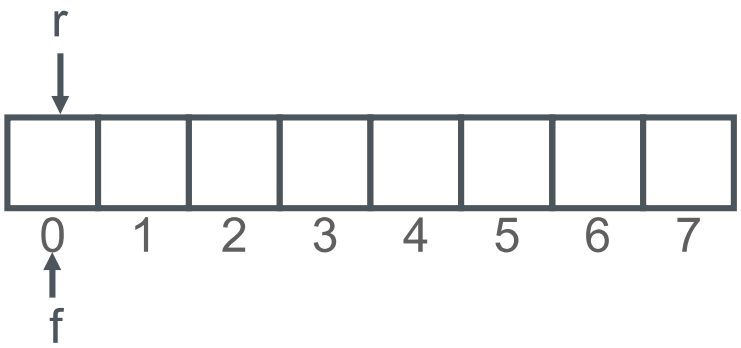
Array-Based Queue - Circular Array

- An array with capacity N stores elements
- **f**: index of front element
- **r**: index of element following rear element
- Each time we increment **f** or **r**, we simply need to compute this increment as:
  - (f + 1) mod N
  - (r + 1) mod N
- modulo operator in C++ is %



Algorithm size():
  return n
Algorithm empty():
  return $(n = 0)$
Algorithm front():
  if empty() **then**
    throw QueueEmpty exception
  return $Q[f]$
Algorithm dequeue():
  if empty() **then**
    throw QueueEmpty exception
  $f \leftarrow (f+1) \bmod N$
  $n = n - 1$
Algorithm enqueue(e):
  if size() $= N$ **then**
    throw QueueFull exception
  $Q[r] \leftarrow e$
  $r \leftarrow (r+1) \bmod N$
  $n = n + 1$

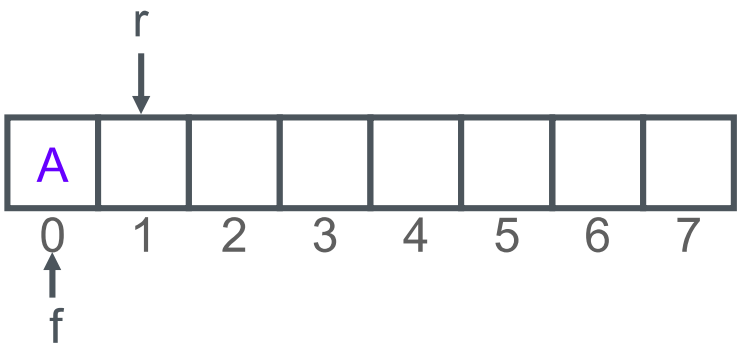
Array-Based Queue - Circular Array



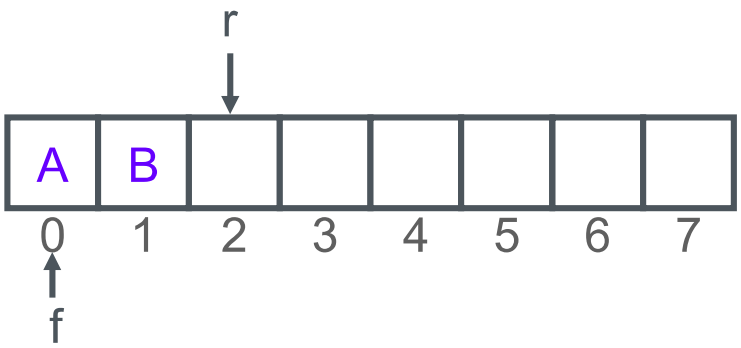
Enq (A)
Enq (B)
Enq (C)
Enq (D)
Deq ()
Deq ()
Deq ()
Enq (E)
Enq (F)
Enq (G)
Enq (H)
Enq (I)

Array-Based Queue - Circular Array

- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)



Array-Based Queue - Circular Array



Enq (A)

Enq (B)

Enq (C)

Enq (D)

Deq ()

Deq ()

Deq ()

Enq (E)

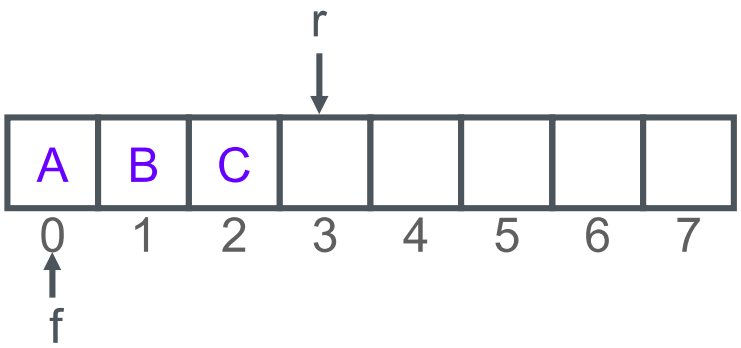
Enq (F)

Enq (G)

Enq (H)

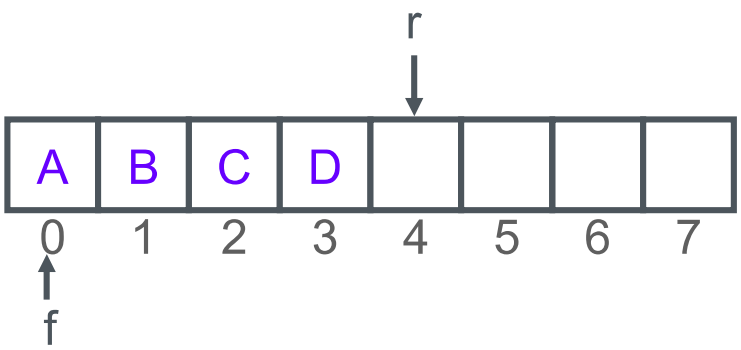
Enq (I)

Array-Based Queue - Circular Array



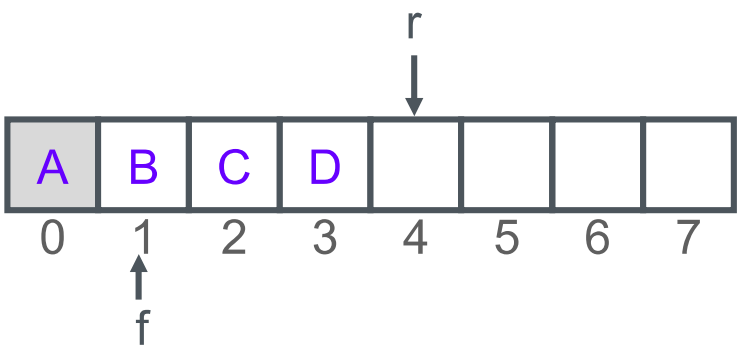
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



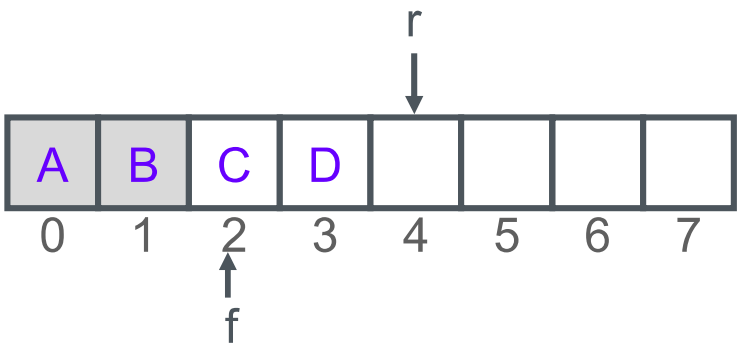
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



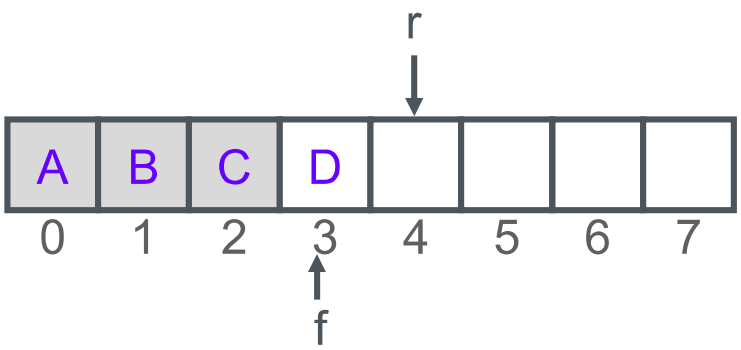
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



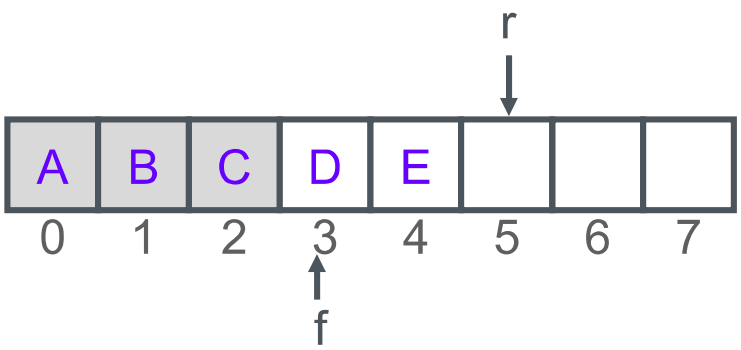
Enq (A)
Enq (B)
Enq (C)
Enq (D)
Deq ()
Deq ()
Deq ()
Enq (E)
Enq (F)
Enq (G)
Enq (H)
Enq (I)

Array-Based Queue - Circular Array



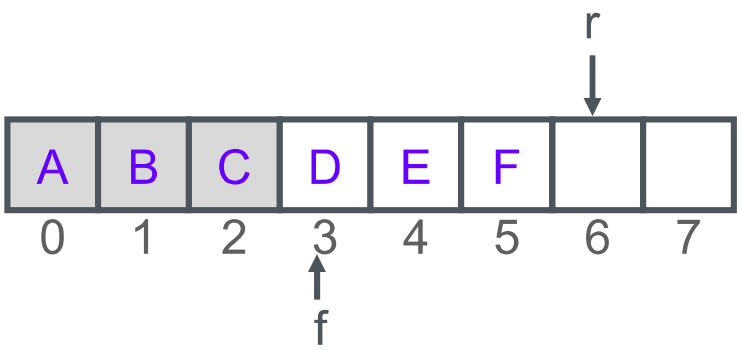
Enq (A)
Enq (B)
Enq (C)
Enq (D)
Deq ()
Deq ()
Deq ()
Enq (E)
Enq (F)
Enq (G)
Enq (H)
Enq (I)

Array-Based Queue - Circular Array



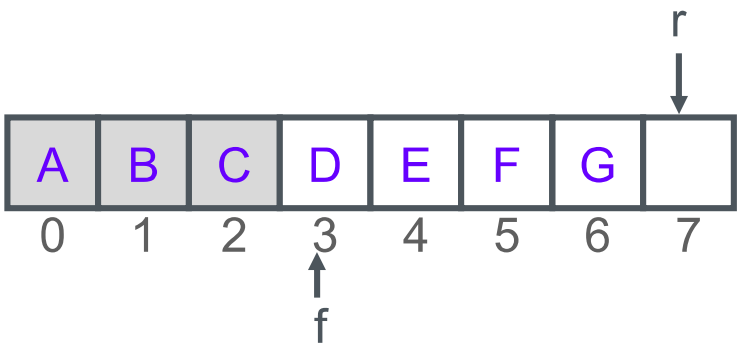
Enq (A)
Enq (B)
Enq (C)
Enq (D)
Deq ()
Deq ()
Deq ()
Enq (E)
Enq (F)
Enq (G)
Enq (H)
Enq (I)

Array-Based Queue - Circular Array



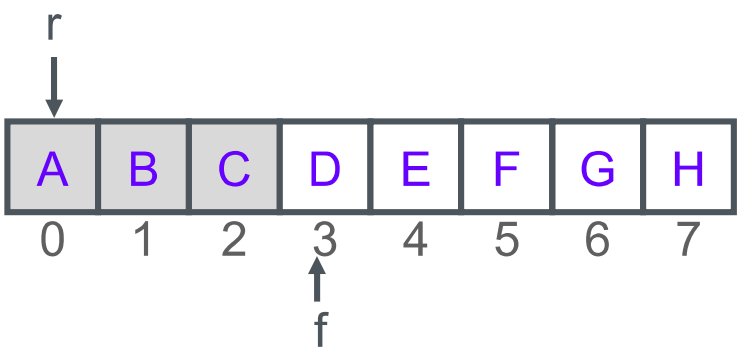
Enq (A)
Enq (B)
Enq (C)
Enq (D)
Deq ()
Deq ()
Deq ()
Enq (E)
Enq (F)
Enq (G)
Enq (H)
Enq (I)

Array-Based Queue - Circular Array



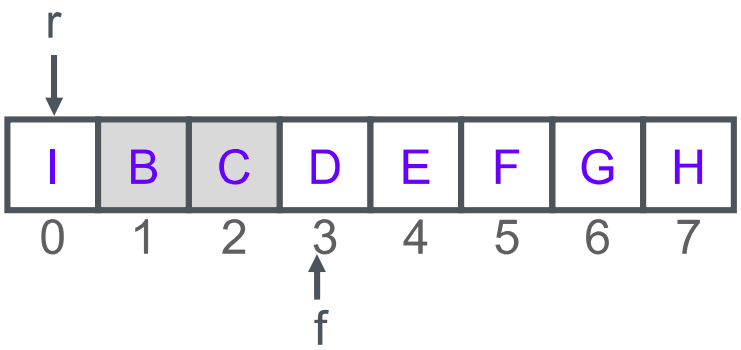
Enq (A)
Enq (B)
Enq (C)
Enq (D)
Deq ()
Deq ()
Deq ()
Enq (E)
Enq (F)
Enq (G)
Enq (H)
Enq (I)

Array-Based Queue - Circular Array



Enq (A)
Enq (B)
Enq (C)
Enq (D)
Deq ()
Deq ()
Deq ()
Enq (E)
Enq (F)
Enq (G)
Enq (H)
Enq (I)

Array-Based Queue - Circular Array



Enq (A)
Enq (B)
Enq (C)
Enq (D)
Deq ()
Deq ()
Deq ()
Enq (E)
Enq (F)
Enq (G)
Enq (H)
Enq (I)

Array-Based Queue - Performance

- Performance:
  - Let **n** be the number of elements in the Queue
  - The space used is **O(N)**
    - this is independent of the number of elements **n**
  - Each operation runs in time **O(1)**

- Limitations:
  - The maximum size of the Queue must be defined beforehand and cannot be changed
  - Trying to enqueue a new element into a full queue causes an implementation-specific exception (QueueFull Exception)

Circularly Linked List-Based Implementation

- We use that CircleList class
  - **C** stores the Queue elements
  - **n** stores the number of elements on Queue

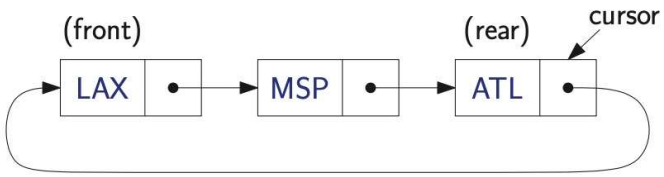
```
typedef string Elem;                          // queue element type
class LinkedQueue {                           // queue as doubly linked list
public:
  LinkedQueue();                              // constructor
  int size() const;                           // number of items in the queue
  bool empty() const;                         // is the queue empty?
  const Elem& front() const throw(QueueEmpty); // the front element
  void enqueue(const Elem& e);                // enqueue element at rear
  void dequeue() throw(QueueEmpty);           // dequeue element at front
private:                                      // member data
  CircleList C;                               // circular list of elements
  int n;                                      // number of elements
};
```

Circularly Linked List-Based Implementation

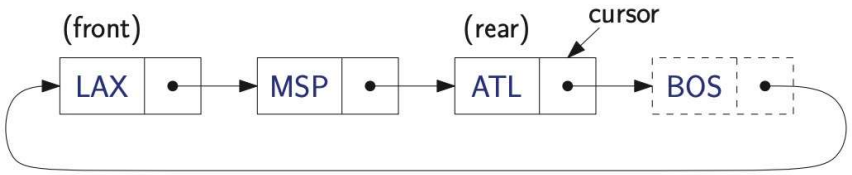
```
LinkedQueue::LinkedQueue()                  // constructor
  : C(), n(0) { }

int LinkedQueue::size() const               // number of items in the queue
  { return n; }

bool LinkedQueue::empty() const             // is the queue empty?
  { return n == 0; }
                                            // get the front element
const Elem& LinkedQueue::front() const throw(QueueEmpty) {
  if (empty())
    throw QueueEmpty("front of empty queue");
  return C.front();                         // list front is queue front
}
```

Circularly Linked List-Based Implementation

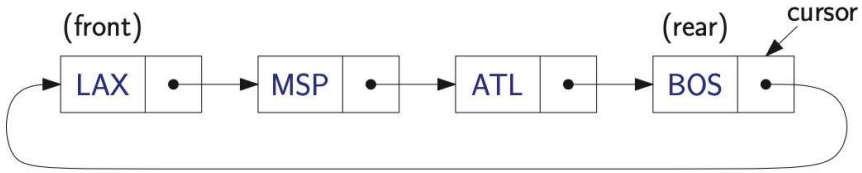
```
                                                    // enqueue element at rear
void LinkedQueue::enqueue(const Elem& e) {
  C.add(e);                                         // insert after cursor
  C.advance();                                      // ...and advance
  n++;
}
```



(a)



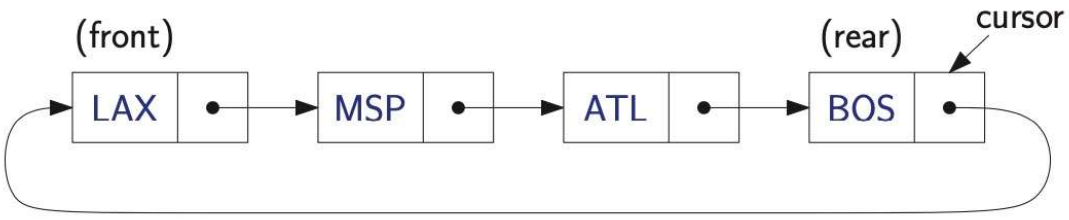
(b)



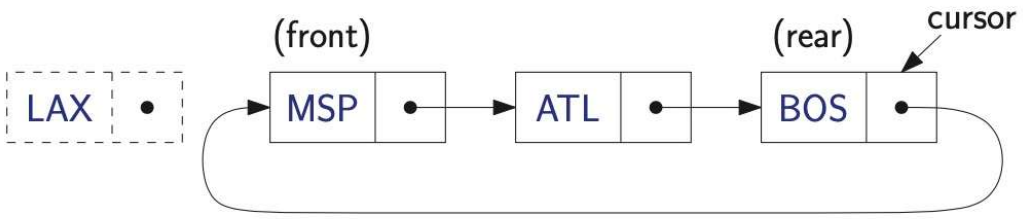
(c)

Circularly Linked List-Based Implementation

```
                                                    // dequeue element at front
void LinkedQueue::dequeue() throw(QueueEmpty) {
  if (empty())
    throw QueueEmpty("dequeue of empty queue");
  C.remove();                                       // remove from list front
  n--;
}
```



(a)



(b)

Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

19 Queues

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

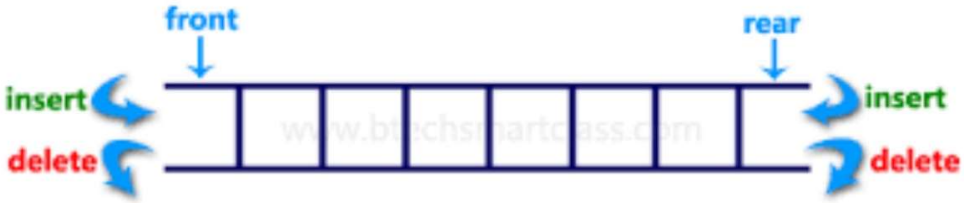
March 7, 2022

Admin.

- One more day for the assignment 2!

Double-Ended Queues

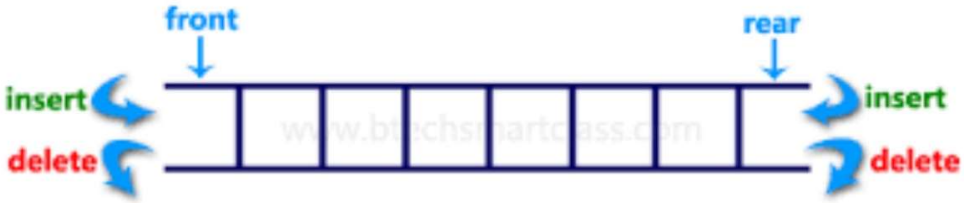
- Double-Ended Queues (sometimes pronounced like ”deck”)
  - supports insertion and deletion at both the front and the rear of the queue



- **insertFront(e):** Insert a new element e at the beginning of the deque.
- **insertBack(e):** Insert a new element e at the end of the deque.
- **eraseFront():** Remove the first element of the deque; an error occurs if the deque is empty.
- **eraseBack():** Remove the last element of the deque; an error occurs if the deque is empty.
- **front():** Return the first element of the deque; an error occurs if the deque is empty.
- **back():** Return the last element of the deque; an error occurs if the deque is empty.
- **size():** Return the number of elements of the deque.
- **empty():** Return true if the deque is empty and false otherwise.

Double-Ended Queues

- Double-Ended Queues (sometimes pronounced like ”deck”)
  - supports insertion and deletion at both the front and the rear of the queue



- A running example:

Operation	*Output*	*D*
insertFront(3)	–	(3)
insertFront(5)	–	(5,3)
front()	*5*	(5,3)
eraseFront()	–	(3)
insertBack(7)	–	(3,7)
back()	*7*	(3,7)
eraseFront()	–	(7)
eraseBack()	–	()

Implementation with Doubly Linked List

- *We will use the functionalities provided by the DLL to implement LinkedDeque's functions. We have seen this pattern a few times before.*

```
typedef string Elem;                          // deque element type
class LinkedDeque {                           // deque as doubly linked list
public:
  LinkedDeque();                              // constructor
  int size() const;                           // number of items in the deque
  bool empty() const;                         // is the deque empty?
  const Elem& front() const throw(DequeEmpty); // the first element
  const Elem& back() const throw(DequeEmpty); // the last element
  void insertFront(const Elem& e);            // insert new first element
  void insertBack(const Elem& e);             // insert new last element
  void removeFront() throw(DequeEmpty);       // remove first element
  void removeBack() throw(DequeEmpty);        // remove last element
private:                                      // member data
  DLinkedList D;                              // linked list of elements
  int n;                                      // number of elements
};
```

Implementation with Doubly Linked List

- We will use the functionalities provided by the DLL to implement LinkedDeque's functions. We have seen this pattern a few times before.
- Performance of a deque realized by a doubly linked list.

Operation	*Time*
size	$O(1)$
empty	$O(1)$
front, back	$O(1)$
insertFront, insertBack	$O(1)$
eraseFront, eraseBack	$O(1)$

- The space used usage is O(n)

```
                                                    // insert new first element
void LinkedDeque::insertFront(const Elem& e) {
  D.addFront(e);
  n++;
}
                                                    // insert new last element
void LinkedDeque::insertBack(const Elem& e) {
  D.addBack(e);
  n++;
}
                                                    // remove first element
void LinkedDeque::removeFront() throw(DequeEmpty) {
  if (empty())
    throw DequeEmpty("removeFront of empty deque");
  D.removeFront();
  n--;
}
                                                    // remove last element
void LinkedDeque::removeBack() throw(DequeEmpty) {
  if (empty())
    throw DequeEmpty("removeBack of empty deque");
  D.removeBack();
  n--;
}
```

Adapter Design Pattern

- Design pattern: which describes a solution to a “typical” software design problem.
  - provides a general template for a solution that can be applied in many different situations.
  - describes the main elements of a solution in an abstract way that can be specialized for a specific problem at hand.
  - In Algorithms:
    - Recursion
    - Using Stack to solve problems
  - In Software Engineering
    - Adapter pattern
    - Iterator pattern
- You remember from previous LinkedDeque, and also Circular Linked List-based implementation of Queue

Adapter Design Pattern

- You remember from previous LinkedDeque, and also Circular Linked List-based implementation of Queue that we took an existing data structure and **adapted** it
  - E.g. we added size **n**
  - We added operations that are meaningful for the new data structure
- For the operations, we have simply **mapped** each deque operation to the corresponding operation of DLinkedList.
- An adapter (also called a wrapper) is a data structure that translates one interface to another.
  - e.g.: In the LinkedDeque implementation:
    - deque operation **insertFront** is mapped to the corresponding operation of DLinkedList **addFront**

Adapter Design Pattern

- Implementing a Stack using Deque:

```
typedef string Elem;                          // element type
class DequeStack {                            // stack as a deque
public:
  DequeStack();                               // constructor
  int size() const;                           // number of elements
  bool empty() const;                         // is the stack empty?
  const Elem& top() const throw(StackEmpty); // the top element
  void push(const Elem& e);                   // push element onto stack
  void pop() throw(StackEmpty);               // pop the stack
private:
  LinkedDeque D;                              // deque of elements
};
```

Stack Method	*Deque Implementation*
size()	size()
empty()	empty()
top()	front()
push(o)	insertFront(o)
pop()	eraseFront()

```
DequeStack::DequeStack()                      // constructor
  : D() { }

                                              // number of elements
int DequeStack::size() const
  { return D.size(); }

                                              // is the stack empty?
bool DequeStack::empty() const
  { return D.empty(); }

                                              // the top element
const Elem& DequeStack::top() const throw(StackEmpty) {
  if (empty())
    throw StackEmpty("top of empty stack");
  return D.front();
}

                                              // push element onto stack
void DequeStack::push(const Elem& e)
  { D.insertFront(e); }

                                              // pop the stack
void DequeStack::pop() throw(StackEmpty)
{
  if (empty())
    throw StackEmpty("pop of empty stack");
  D.removeFront();
}
```

Adapter Design Pattern

- Implementing a Queue using Deque:

Queue Method	*Deque Implementation*
size()	size()
empty()	empty()
front()	front()
enqueue(e)	insertBack(e)
dequeue()	eraseFront()

- The operations are equally efficient.
- We have used and will use this design pattern many times.

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - string (String class with all operations)
  - stack (Container with last-in, first-out access)
  - queue (Container with first-in, first-out access)
  - deque (Double-ended queue)
  - vector (Resizable array)
  - list (Doubly linked list)
  - priority queue (Queue ordered by value)
  - set (Set)
  - map Associative array (dictionary)

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - string:

s.find(p)	Return the index of first occurrence of string *p* in *s*
s.find(p, i)	Return the index of first occurrence of string *p* in *s* on or after position *i*
s.substr(i,m)	Return the substring starting at position *i* of *s* and consisting of *m* characters
s.insert(i, p)	Insert string *p* just prior to index *i* in *s*
s.erase(i, m)	Remove the substring of length *m* starting at index *i*
s.replace(i, m, p)	Replace the substring of length *m* starting at index *i* with *p*
getline(is, s)	Read a single line from the input stream *is* and store the result in *s*

```
#include <string>
using std::string;
// ...
string s = "to be";
string t = "not " + s;              // t = "not to be"
string u = s + " or " + t;          // u = "to be or not to be"
if (s > t)                          // true: "to be" > "not to be"
  cout << u;                        // outputs "to be or not to be"


string s = "John";                  // s = "John"
int i = s.size();                   // i = 4
char c = s[3];                      // c = 'n'
s += " Smith";                      // now s = "John Smith"
```

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - vector:

```
#include <vector>
using namespace std;                    // make std accessible

vector<int> scores(100);                // 100 integer scores
vector<char> buffer(500);               // buffer of 500 characters
vector<Passenger> passenList(20);       // list of 20 Passengers
```

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - stack:

```
#include <stack>
using std::stack;                    // make stack accessible
stack<int> myStack;                  // a stack of integers
```

size():	Return the number of elements in the stack.
empty():	Return true if the stack is empty and false otherwise.
push(e):	Push e onto the top of the stack.
pop():	Pop the element at the top of the stack.
top():	Return a reference to the element at the top of the stack.

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - queue:

```
#include <queue>
using std::queue;                    // make queue accessible
queue<float> myQueue;                // a queue of floats
```

size():	Return the number of elements in the queue.
empty():	Return true if the queue is empty and false otherwise.
push(*e*):	Enqueue *e* at the rear of the queue.
pop():	Dequeue the element at the front of the queue.
front():	Return a reference to the element at the queue's front.
back():	Return a reference to the element at the queue's rear.

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - deque:

```
#include <deque>
using std::deque;                    // make deque accessible
deque<string> myDeque;               // a deque of strings
```

size():	Return the number of elements in the deque.
empty():	Return true if the deque is empty and false otherwise.
push_front(e):	Insert e at the beginning the deque.
push_back(e):	Insert e at the end of the deque.
pop_front():	Remove the first element of the deque.
pop_back():	Remove the last element of the deque.
front():	Return a reference to the deque's first element.
back():	Return a reference to the deque's last element.

List and Sequence Containers

- Vector (also called Array List)
  - Access each element using a notion of index in [0,n-1]
  - Index of element e: the number of elements that are before e
  - Typically we use the “index” (e.g., [ ])
  - A more general ADT than “array”
- List
  - Not using an index to access, but use a node to access
  - Insert a new element e before some “position” p
  - A more general ADT than “linked list”
- Sequence
  - Can access an element as vector and list (using both index and position)

The Vector ADT

- The Vector or Array List ADT extends the notion of array by storing a sequence of objects
- An element can be accessed, inserted or removed by specifying its index (number of elements preceding it)
- An exception is thrown if an incorrect index is given (e.g., a negative index)
- Main methods:

$\mathsf{at}(i)$: Return the element of V with index i; an error condition occurs if i is out of range.

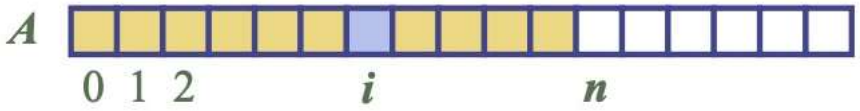
$\mathsf{set}(i, e)$: Replace the element at index i with e; an error condition occurs if i is out of range.

$\mathsf{insert}(i, e)$: Insert a new element e into V to have index i; an error condition occurs if i is out of range.

$\mathsf{erase}(i)$: Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation **at(i)** is implemented in O(1) time by returning A[i]
- Operation set(i,o) is implemented in O(1) time by performing A[i] = o



at(i): Return the element of V with index i; an error condition occurs if i is out of range.

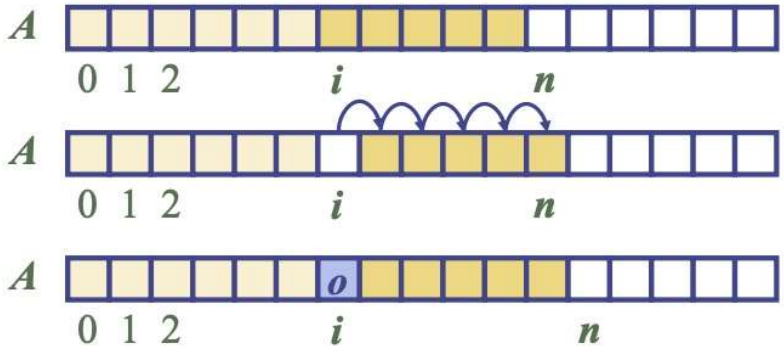
set(i, e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i, e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector - Insertion

- In operation insert(i, o), we need to make room for the new element by shifting forward the n - i elements A[i], …, A[n - 1]
  - In the worst case (i = 0), this takes O(n) time



at(i): Return the element of V with index i; an error condition occurs if i is out of range.

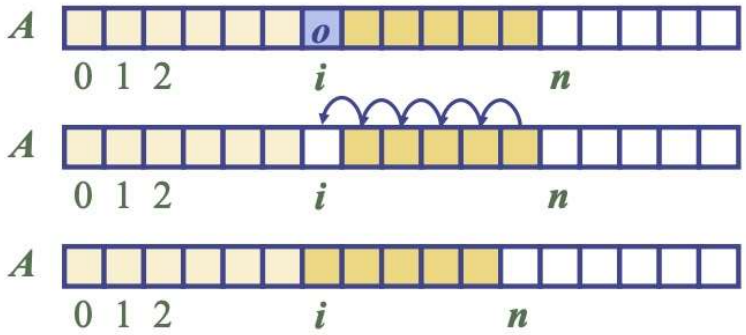
set(i, e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i, e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector - Removal

- In operation erase(i), we need to fill the hole left by the removed element by shifting backward the n - i - 1 elements A[i + 1], …, A[n - 1]
  - In the worst case (i = 0), this takes O(n) time



at(i): Return the element of V with index i; an error condition occurs if i is out of range.

set(i, e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i, e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector - Performance

- In the array-based implementation of an array list:
  - The space used by the data structure is O(n)
  - size, empty, at and set run in O(1) time
  - insert and erase run in O(n) time in worst case
- If we use the array in a circular fashion, operations insert(0, x) and erase(0, x) run in O(1) time
- In an insert operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Operation	*Time*
size()	$O(1)$
empty()	$O(1)$
at(i)	$O(1)$
set(i, e)	$O(1)$
insert(i, e)	$O(n)$
erase(i)	$O(n)$

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n insert(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an insert operation the average time taken by an insert over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n insert operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

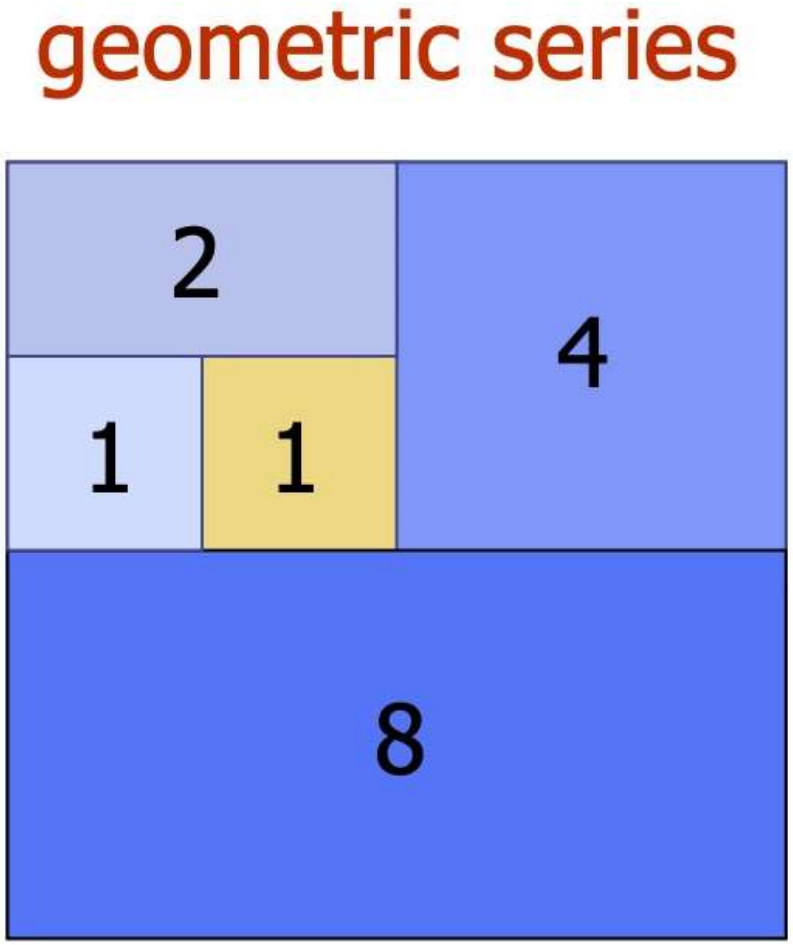
$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an insert operation is $O(n)$

Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time T(n) of a series of n insert operations is proportional to
- $n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + 2^{k+1} - 1 =$
- $3n - 1$
- T(n) is O(n)
- The amortized time of an insert operation is O(1)



Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

21 Containers

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

March 10, 2022

List and Sequence Containers

- Vector (also called Array List)
  - Access each element using a notion of index in [0,n-1]
  - Index of element e: the number of elements that are before e
  - Typically we use the “index” (e.g., [ ])
  - A more general ADT than “array”
- List
  - Not using an index to access, but use a node to access
  - Insert a new element e before some “position” p
  - A more general ADT than “linked list”
- Sequence
  - Can access an element as vector and list (using both index and position)

The Vector ADT

- The Vector or Array List ADT extends the notion of array by storing a sequence of objects
- An element can be accessed, inserted or removed by specifying its index (number of elements preceding it)
- An exception is thrown if an incorrect index is given (e.g., a negative index)
- Main methods:

$\mathsf{at}(i)$: Return the element of V with index i; an error condition occurs if i is out of range.

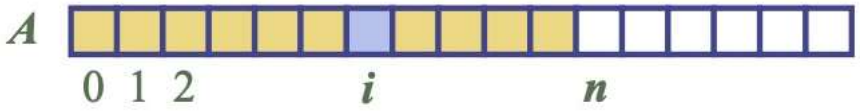
$\mathsf{set}(i, e)$: Replace the element at index i with e; an error condition occurs if i is out of range.

$\mathsf{insert}(i, e)$: Insert a new element e into V to have index i; an error condition occurs if i is out of range.

$\mathsf{erase}(i)$: Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation **at(i)** is implemented in O(1) time by returning A[i]
- Operation set(i,o) is implemented in O(1) time by performing A[i] = o



at(i): Return the element of V with index i; an error condition occurs if i is out of range.

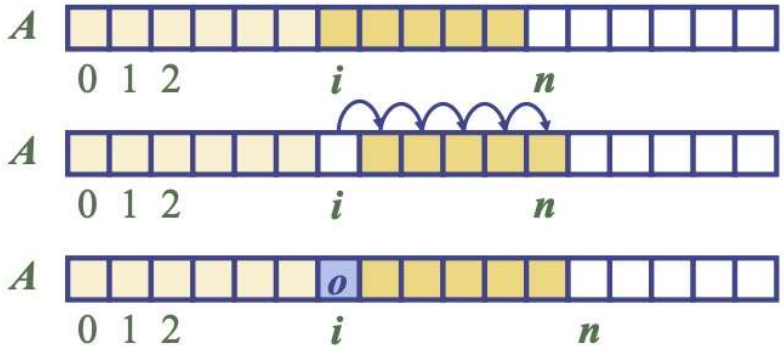
set(i, e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i, e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector - Insertion

- In operation insert(i, o), we need to make room for the new element by shifting forward the n - i elements A[i], …, A[n - 1]
  - In the worst case (i = 0), this takes O(n) time



at(i): Return the element of V with index i; an error condition occurs if i is out of range.

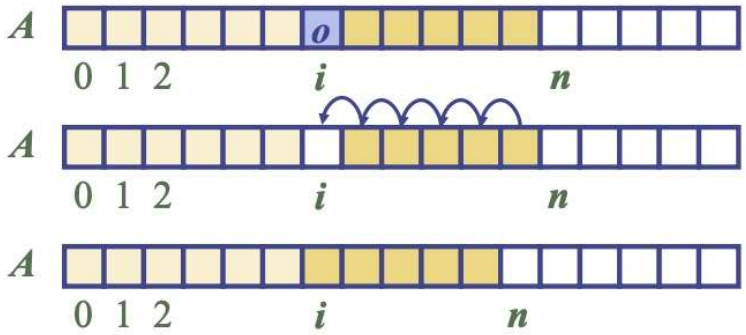
set(i, e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i, e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector - Removal

- In operation erase(i), we need to fill the hole left by the removed element by shifting backward the n - i - 1 elements A[i + 1], …, A[n - 1]
  - In the worst case (i = 0), this takes O(n) time



at(i): Return the element of V with index i; an error condition occurs if i is out of range.

set(i, e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i, e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector in C++

```
typedef int Elem;                                  // base element type
class ArrayVector {
public:
  ArrayVector();                                   // constructor
  int size() const;                                // number of elements
  bool empty() const;                              // is vector empty?
  Elem& operator[](int i);                         // element at index
  Elem& at(int i) throw(IndexOutOfBounds); // element at index
  void erase(int i);                               // remove element at index
  void insert(int i, const Elem& e);               // insert element at index
  void reserve(int N);                             // reserve at least N spots
  // ... (housekeeping functions omitted)
private:
  int capacity;                                    // current array size
  int n;                                           // number of elements in vector
  Elem* A;                                         // array storing the elements
};
```

Array-based Implementation of Vector in C++

```
ArrayVector::ArrayVector()                    // constructor
  : capacity(0), n(0), A(NULL) { }

int ArrayVector::size() const                 // number of elements
  { return n; }

bool ArrayVector::empty() const               // is vector empty?
  { return size() == 0; }

Elem& ArrayVector::operator[](int i)          // element at index
  { return A[i]; }
                                              // element at index (safe)
Elem& ArrayVector::at(int i) throw(IndexOutOfBounds) {
  if (i < 0 || i >= n)
    throw IndexOutOfBounds("illegal index in function at()");
  return A[i];
}

void ArrayVector::erase(int i) {              // remove element at index
  for (int j = i+1; j < n; j++)               // shift elements down
    A[j - 1] = A[j];
  n--;                                        // one fewer element
}
```

Array-based Implementation of Vector in C++

- The reserve function first checks whether the capacity already exceeds ***n***, in which case nothing needs to be done.
- The insert function first checks whether there is sufficient capacity for one more element. If not, it sets the capacity to the maximum of 1 and twice the current capacity.

```
void ArrayVector::reserve(int N) {          // reserve at least N spots
  if (capacity >= N) return;                // already big enough
  Elem* B = new Elem[N];                    // allocate bigger array
  for (int j = 0; j < n; j++)               // copy contents to new array
    B[j] = A[j];
  if (A != NULL) delete [] A;               // discard old array
  A = B;                                    // make B the new array
  capacity = N;                             // set new capacity
}
void ArrayVector::insert(int i, const Elem& e) {
  if (n >= capacity)                        // overflow?
    reserve(max(1, 2 * capacity));          // double array size
  for (int j = n - 1; j >= i; j--)          // shift elements up
    A[j+1] = A[j];
  A[i] = e;                                 // put in empty slot
  n++;                                      // one more element
}
```

Array-based Implementation of Vector - Performance

- In the array-based implementation of an array list:
  - The space used by the data structure is O(n)
  - size, empty, at and set run in O(1) time
  - insert and erase run in O(n) time in worst case
- If we use the array in a circular fashion, operations insert(0, x) and erase(0, x) run in O(1) time
- In an insert operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Operation	*Time*
size()	$O(1)$
empty()	$O(1)$
at(i)	$O(1)$
set(i, e)	$O(1)$
insert(i, e)	$O(n)$
erase(i)	$O(n)$

Comparison of the Strategies to Resize Array

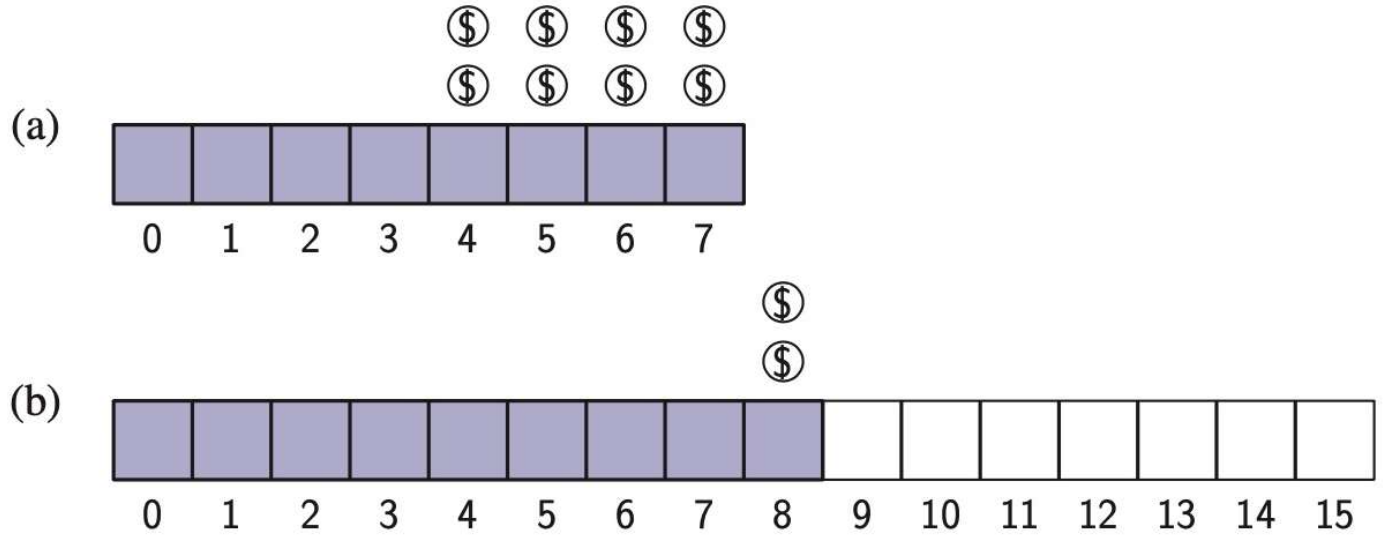
- Suppose we are doing only push operations
- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n insert(o) operations
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

```
Algorithm insert(o)
  if t = S.length − 1 then
    A ← new array of
          size …
    for i ← 0 to n−1 do
      A[i] ← S[i]
    S ← A
  n ← n + 1
  S[n−1] ← o
```

- We assume that we start with an empty array of size 2
- We call **amortized time** of an insert operation the average time taken by an insert over the series of operations, i.e., $T(n)/n$

Comparison of the Strategies to Resize Array

- Amortizations:
  - Certain operations may be extremely costly
  - But they cannot occur frequently enough to slow down the entire program
    - The less costly operations far outnumber the costly one
    - Thus, over the long term they are “paying back”



Comparison of the Strategies to Resize Array

- Amortizations:
  - Certain operations may be extremely costly
  - But they cannot occur frequently enough to slow down the entire program
    - The less costly operations far outnumber the costly one
    - Thus, over the long term they are “paying back”
  - The idea:
    - The worst-case operation can alter the state in such a way that “the worst case cannot occur again for a long time.
      - Thus, amortizing its cost

Comparison of the Strategies to Resize Array

- Suppose we are doing only push operations
- total time $T(n)$: to perform a series of n insert(o) operations
- Remember that every push (storing an element) takes 1 unit of time. After all we will have n pushes.
- Each array resize needs a time proportional to the size of the old array
- What is the time for n push operation
- For simplicity, we start with an array of capacity 2 and size zero and grow it dynamically
- We have to identify how many times the array resizes
- We call amortized time of an insert operation the average time taken by an insert over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
  - Suppose if you want to increment its size by 2, adding two more spaces.
- We suppose c = 2. There will be k=n/2 array resizes.
- For the incremental approach, each time we go past our capacity (k=(n/c)=(n/2)) times, we will increase capacity by **c=2** and we will have to copy the stuff already in the array into the new array.
- Assuming each item we copy requires 1 time unit.
- For 2 items: 2 units of time

  For 4 items: 4 units of time

  For 6 items: 6 units of time
- We the have need 2 + 4 + 6 + 8 + … + 2*k units of time
- Total time = n + 2 + 4 + 6 + 8 + … + 2*k = n + c(1+2+... + k).
- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an insert operation is $O(n^2)$ /n which is $O(n)$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
  - Suppose if you want to increment its size by 2, adding two more spaces. There will be k=n/2 array resizes.

- The total time $T(n)$ of a series of n insert operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

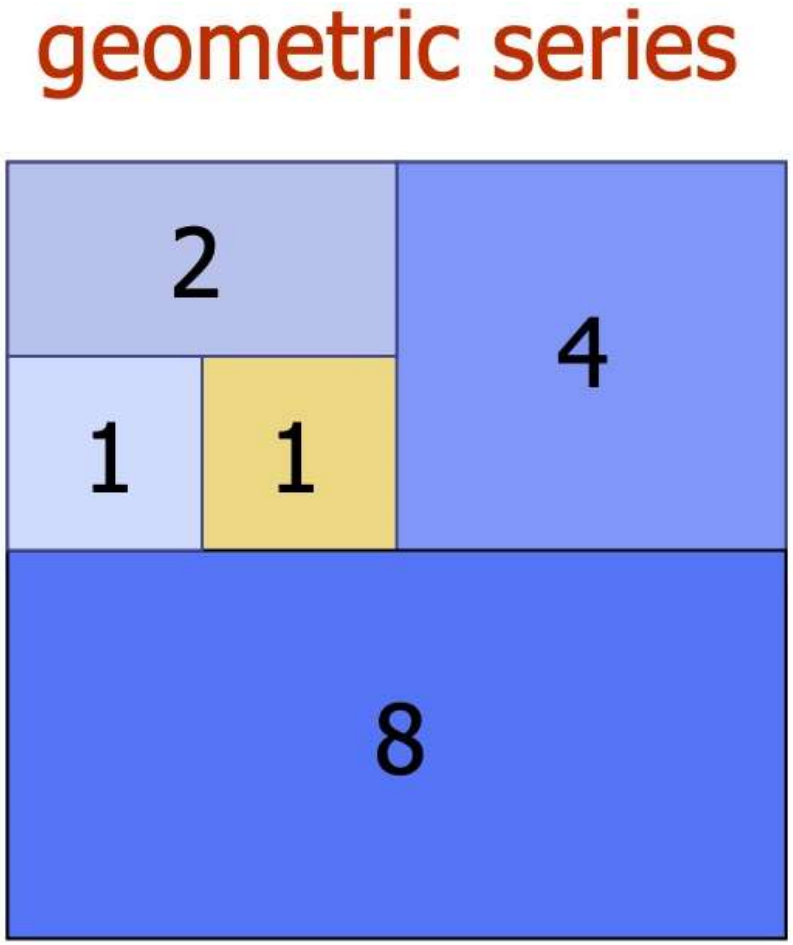
$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an insert operation is $O(n)$

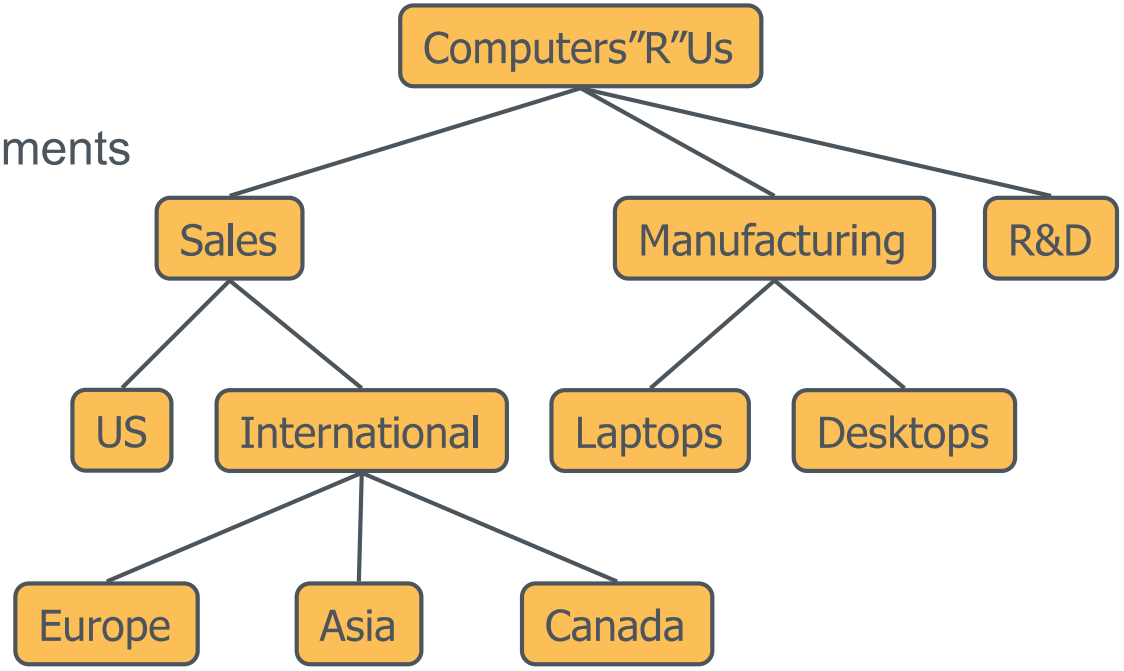
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time T(n) of a series of n insert operations is proportional to
- $n + 2 + 4 + 8 + \ldots + 2^k =$
  $n + 2^{k+1} - 1 =$
- $3n - 1$
- T(n) is O(n)
- The amortized time of an insert operation is O(1)



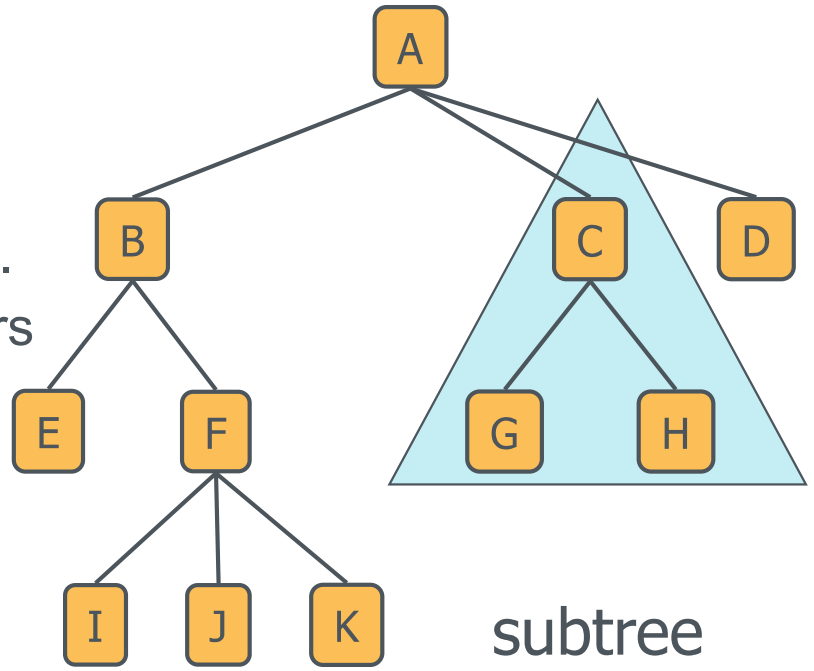
Trees

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Trees

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.



- Subtree: tree consisting of a node and its descendants

Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

22 Lists, Positions, and Trees

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

March 14, 2022

Admin

- Mid-Term 2:
  - Wednesday 23 March 2022
  - Duration: **1 hour**
  - Location: TBA
  - Covers: Topics from “Doubly Linked Lists” until the lecture of Wednesday 16 March 2022 (inclusive)

Array-based Implementation of Vector in C++ - Review

- The reserve function first checks whether the capacity already exceeds ***n***, in which case nothing needs to be done.
- The insert function first checks whether there is sufficient capacity for one more element. If not, it sets the capacity to the maximum of 1 and twice the current capacity.

```
void ArrayVector::reserve(int N) {          // reserve at least N spots
  if (capacity >= N) return;                // already big enough
  Elem* B = new Elem[N];                    // allocate bigger array
  for (int j = 0; j < n; j++)               // copy contents to new array
    B[j] = A[j];
  if (A != NULL) delete [] A;               // discard old array
  A = B;                                    // make B the new array
  capacity = N;                             // set new capacity
}
void ArrayVector::insert(int i, const Elem& e) {
  if (n >= capacity)                        // overflow?
    reserve(max(1, 2 * capacity));          // double array size
  for (int j = n - 1; j >= i; j--)          // shift elements up
    A[j+1] = A[j];
  A[i] = e;                                 // put in empty slot
  n++;                                      // one more element
}
```

Comparison of the Strategies to Resize Array

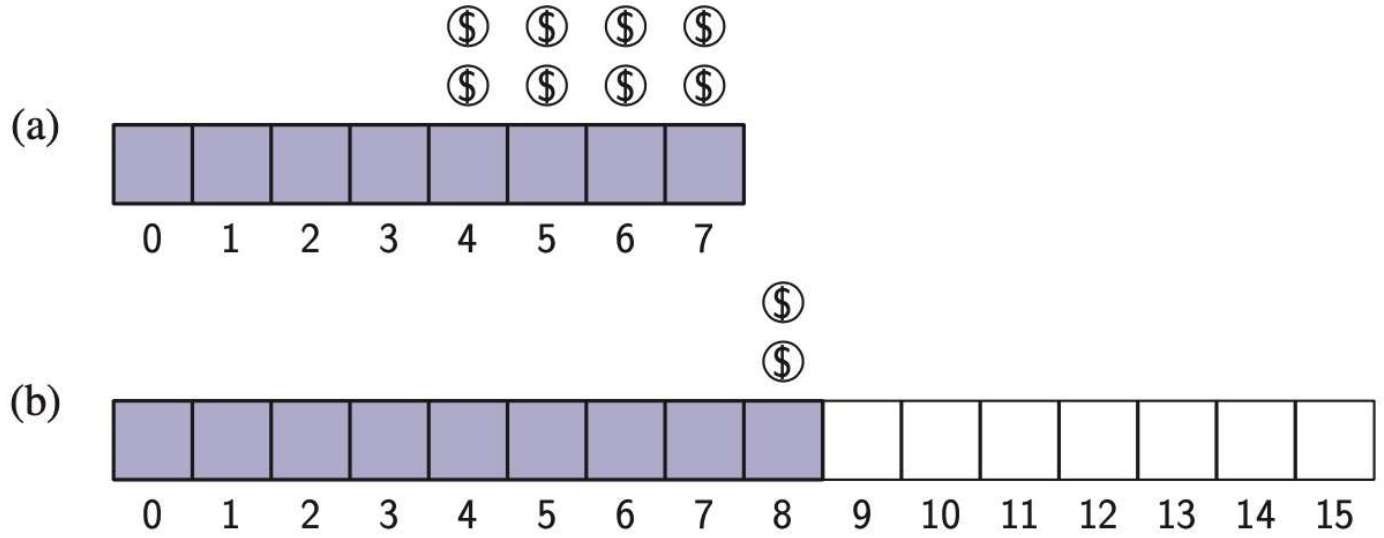
- Suppose we are doing only push operations
- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n insert(o) operations
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

```
Algorithm insert(o)
  if t = S.length − 1 then
    A ← new array of
          size …
    for i ← 0 to n−1 do
      A[i] ← S[i]
    S ← A
  n ← n + 1
  S[n−1] ← o
```

- We assume that we start with an empty array of size 2
- We call **amortized time** of an insert operation the average time taken by an insert over the series of operations, i.e., $T(n)/n$

Comparison of the Strategies to Resize Array

- Amortizations:
  - Certain operations may be extremely costly
  - But they cannot occur frequently enough to slow down the entire program
    - The less costly operations far outnumber the costly one
    - Thus, over the long term they are “paying back”



Comparison of the Strategies to Resize Array

- Amortizations:
  - Certain operations may be extremely costly
  - But they cannot occur frequently enough to slow down the entire program
    - The less costly operations far outnumber the costly one
    - Thus, over the long term they are “paying back”
  - The idea:
    - The worst-case operation can alter the state in such a way that “the worst case cannot occur again for a long time.
      - Thus, amortizing its cost

Comparison of the Strategies to Resize Array

- Suppose we are doing only push operations
- total time $T(n)$: to perform a series of n insert(o) operations
- Remember that every push (storing an element) takes 1 unit of time. After all we will have n pushes.
- Each array resize needs a time proportional to the size of the old array
- What is the time for n push operation
- For simplicity, we start with an array of capacity 2 and size zero and grow it dynamically
- We have to identify how many times the array resizes
- We call amortized time of an insert operation the average time taken by an insert over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
  - Suppose if you want to increment its size by 2, adding two more spaces.
- We suppose c = 2. There will be k=n/2 array resizes.
- For the incremental approach, each time we go past our capacity (k=(n/c)=(n/2)) times, we will increase capacity by **c=2** and we will have to copy the stuff already in the array into the new array.
- Assuming each item we copy requires 1 time unit.
- For 2 items: 2 units of time

  For 4 items: 4 units of time

  For 6 items: 6 units of time
- We the have need 2 + 4 + 6 + 8 + … + 2*k units of time
- Total time = n + 2 + 4 + 6 + 8 + … + 2*k = n + c(1+2+... + k).
- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an insert operation is $O(n^2)$ /n which is $O(n)$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
  - Suppose if you want to increment its size by 2, adding two more spaces. There will be k=n/2 array resizes.

- The total time $T(n)$ of a series of n insert operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

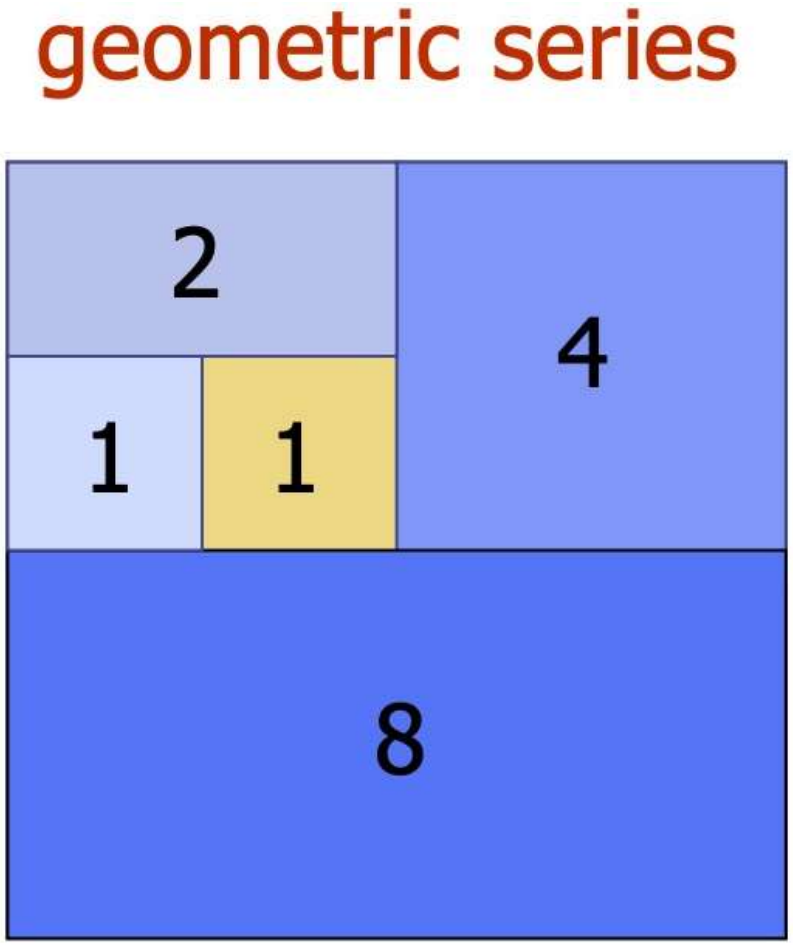
$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an insert operation is $O(n)$

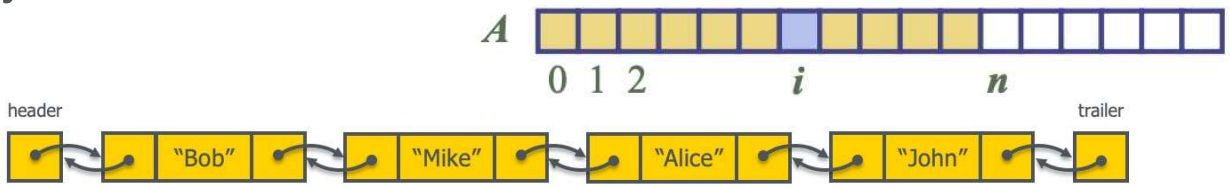
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time T(n) of a series of n insert operations is proportional to
- $n + 2 + 4 + 8 + \ldots + 2^k =$
  $n + 2^{k+1} - 1 =$
- $3n - 1$
- T(n) is O(n)
- The amortized time of an insert operation is O(1)



Linear Data Structures and Positions

- We have seen many data structures
  - Vector
  - Linked-Lists



- Position: Where a data element is stored
- The **Position ADT** models the notion of place within a data structure where a single object is stored
  - abstracts the notion of the relative position or place of an element within a list
- It gives a unified view of diverse ways of storing data, such as:
  - a cell of an array
  - a node of a linked list
- Just one method:
  - *object* p.element(): returns the element at position
  - In C++ it is convenient to implement this as *p

(Node) List ADT

- The **Node List ADT** models
  - a sequence of positions storing objects
  - It establishes a **before/after** relation between positions
  - Generic methods:
    - size(), empty()
  - Iterators:
    - begin(), end()
  - Update methods:
    - insertFront(e), insertBack(e), removeFront(), removeBack()
  - Iterator-based update:
    - insert(p, e), remove(p)

(Node) List ADT

- The **Node List ADT** models
  - a sequence of positions storing objects
  - It establishes a **before/after** relation between positions
  - Generic methods:
    - size(), empty()
  - Iterators:
    - begin(), end()
  - Update methods:
    - insertFront(e), insertBack(e), removeFront(), removeBack()
  - Iterator-based update:
    - insert(p, e), remove(p)
  - **No method for accessing a specific node?**

(Node) List ADT

- This implementation is basically a Doubly Linked-List
- **n**:
  - The number of data elements
- Iterator is an implementation of the Position ADT
  - We use iterator instead of a pointer to a specific node in this implementation

```
typedef int Elem;                                  // list base element type
class NodeList {                                   // node-based list
private:
  // insert Node declaration here...
public:
  // insert Iterator declaration here...
public:
  NodeList();                                      // default constructor
  int size() const;                                // list size
  bool empty() const;                              // is the list empty?
  Iterator begin() const;                          // beginning position
  Iterator end() const;                            // (just beyond) last position
  void insertFront(const Elem& e);                 // insert at front
  void insertBack(const Elem& e);                  // insert at rear
  void insert(const Iterator& p, const Elem& e);   // insert e before p
  void eraseFront();                               // remove first
  void eraseBack();                                // remove last
  void erase(const Iterator& p);                   // remove p
  // housekeeping functions omitted...
private:                                           // data members
  int      n;                                      // number of items
  Node*  header;                                   // head-of-list sentinel
  Node*  trailer;                                  // tail-of-list sentinel
};
```

Containers and Iterators

- An **iterator** abstracts the process of scanning through a collection of elements
- A **container** is an abstract data structure that supports element access through iterators
  - Examples include Stack, Queue, Vector, List
  - **begin():** returns an iterator to the first element
  - **end():** return an iterator to an imaginary position just after the last element
- An iterator behaves like a pointer to an element
  - ***p:** returns the element referenced by this iterator
  - **++p:** advances to the next element
  - Iterator extends the concept of **position** by adding a traversal capability

Iterating through a Container

- Let **C** be a **container** and **p** be an **iterator** for **C**

```
for (p = C.begin(); p != C.end(); ++p)
    loop_body
```

- Example: (with an STL vector)
  - Notice how for loop is implemented
  - Notice how we access the element of iterator

```
Output:
Iterator is on data 1
Iterator is on data 2
Iterator is on data 3
Iterator is on data 4
Iterator is on data 5
The sum is :15
```

```cpp
#include <iostream>
#include <vector>
using namespace std;

int  main(){
    vector<int> v1;
    for (int i = 1; i <= 5; i++)
        v1.push_back(i);

    typedef vector<int>::iterator Iterator;
    int sum = 0;
    for (Iterator p = v1.begin(); p != v1.end(); ++p){
        cout << "Iterator is on data " << *p << endl;
        sum += *p;
    }
    cout << "The sum is :" << sum << endl;

}
```

Implementing Iterators

- Array-based
  - **Array A of the n elements**
  - **index i** that keeps track of the cursor
  - **begin()** = 0
  - **end()** = n (index following the last element)
- Linked list-based
  - For example: A doubly linked-list **L** storing the elements, with sentinels for header and trailer
  - **pointer to node** containing the current element
  - **begin()** = front node (immediately after the header)
  - **end()** = trailer node (immediately after last node)

Implementation of Iterator for (Node) List

- This is an implementation of the iterator for our NodeList (aka DLL)

```
class Iterator {                                  // an iterator for the list
public:
  Elem& operator*();                              // reference to the element
  bool operator==(const Iterator& p) const; // compare positions
  bool operator!=(const Iterator& p) const;
  Iterator& operator++();                         // move to next position
  Iterator& operator--();                         // move to previous position
  friend class NodeList;                          // give NodeList access
private:
  Node* v;                                        // pointer to the node
  Iterator(Node* u);                              // create from node
};
```

```
NodeList::Iterator::Iterator(Node* u)             // constructor from Node*
  { v = u; }

Elem& NodeList::Iterator::operator*()             // reference to the element
  { return v->elem; }
                                                  // compare positions
bool NodeList::Iterator::operator==(const Iterator& p) const
  { return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const
  { return v != p.v; }
                                                  // move to next position
NodeList::Iterator& NodeList::Iterator::operator++()
  { v = v->next; return *this; }
                                                  // move to previous position
NodeList::Iterator& NodeList::Iterator::operator--()
  { v = v->prev; return *this; }
```

```
struct Node {
  Elem elem;
  Node* prev;
  Node* next;
};
```

(Node) List ADT (duplicate slide)

- This implementation is basically a Doubly Linked-List
- **n**:
  - The number of data elements
- Iterator is an implementation of the Position ADT
  - We use iterator instead of a pointer to a specific node in this implementation

```
typedef int Elem;                                  // list base element type
class NodeList {                                   // node-based list
private:
  // insert Node declaration here...
public:
  // insert Iterator declaration here...
public:
  NodeList();                                      // default constructor
  int size() const;                                // list size
  bool empty() const;                              // is the list empty?
  Iterator begin() const;                          // beginning position
  Iterator end() const;                            // (just beyond) last position
  void insertFront(const Elem& e);                 // insert at front
  void insertBack(const Elem& e);                  // insert at rear
  void insert(const Iterator& p, const Elem& e); // insert e before p
  void eraseFront();                               // remove first
  void eraseBack();                                // remove last
  void erase(const Iterator& p);                   // remove p
  // housekeeping functions omitted...
private:                                           // data members
  int     n;                                       // number of items
  Node*  header;                                   // head-of-list sentinel
  Node*  trailer;                                  // tail-of-list sentinel
};
```

STL Iterators in C++

- Each STL container type C supports iterators:
  - **C::iterator** – read/write iterator type
  - **C::const_iterator** – read-only iterator type
  - **C.begin(), C.end()** – return start/end iterators
- This iterator-based operators and methods:
  - ***p:** access current element
  - **++p, --p:** advance to next/previous element
  - **C.assign(p, q):** replace C with contents referenced by the iterator range [p, q) (from p up to, but not including, q)
  - **insert(p, e):** insert e prior to position p
  - **erase(p):** remove element at position p
  - **erase(p, q):** remove elements in the iterator range [p, q)

STL List in C++

```
#include <list>
using std::list;                    // make list accessible
list<float> myList;                 // an empty list of floats
```

list(n): Construct a list with n elements; if no argument list is given, an empty list is created.

size(): Return the number of elements in L.

empty(): Return true if L is empty and false otherwise.

front(): Return a reference to the first element of L.

back(): Return a reference to the last element of L.

push_front(e): Insert a copy of e at the beginning of L.

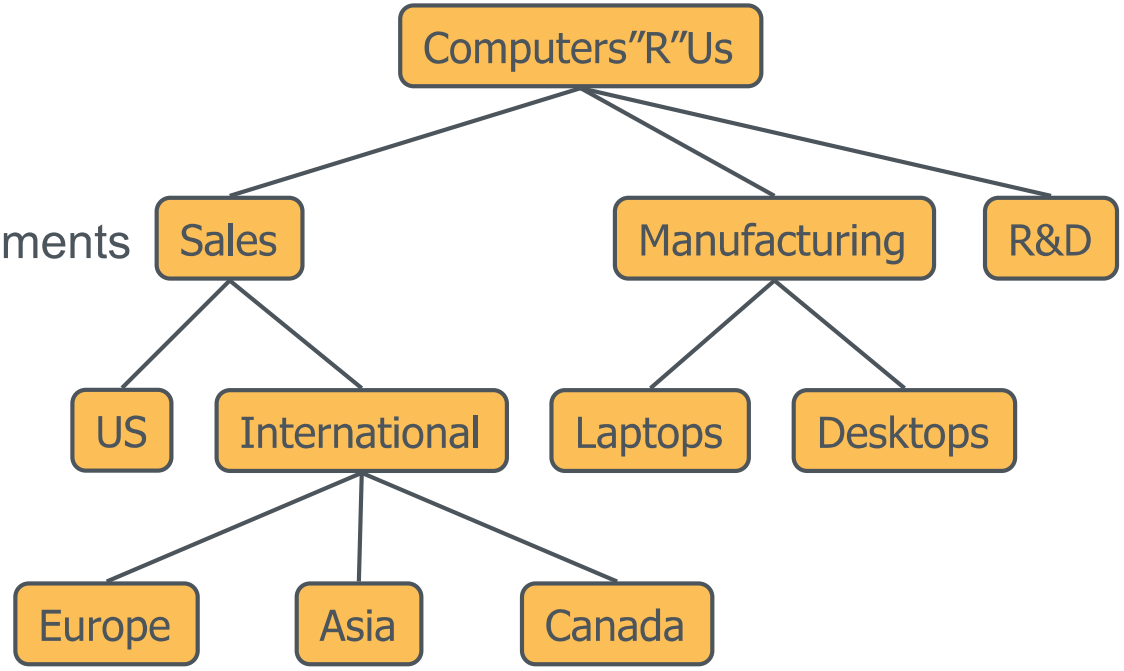
push_back(e): Insert a copy of e at the end of L.

pop_front(): Remove the fist element of L.

pop_back(): Remove the last element of L.

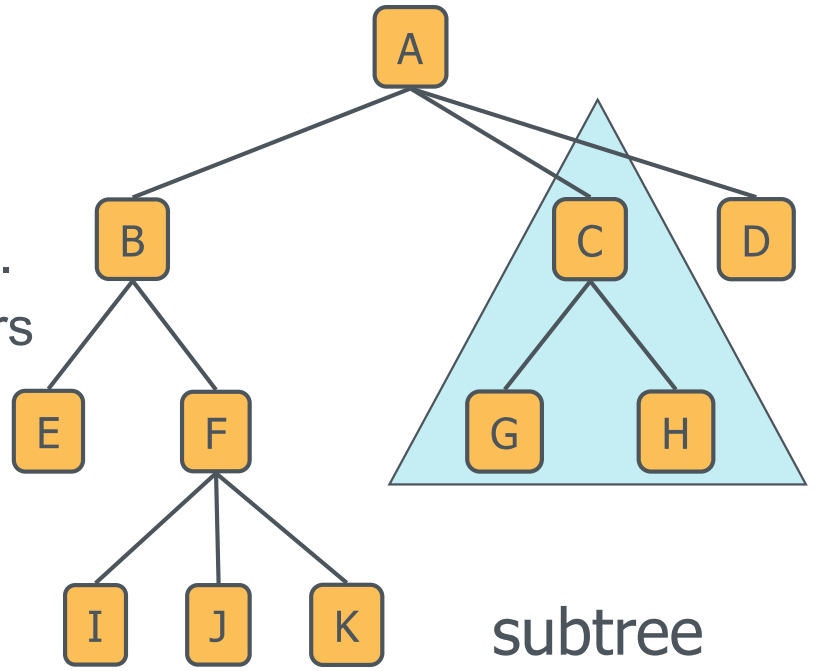
Trees

- In computer science, a tree is an abstract model of a hierarchical structure
- The relation between elements is non-linear
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Trees

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.



- Subtree: tree consisting of a node and its descendants

Questions?

MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

23 Trees

Department of Computing and Software

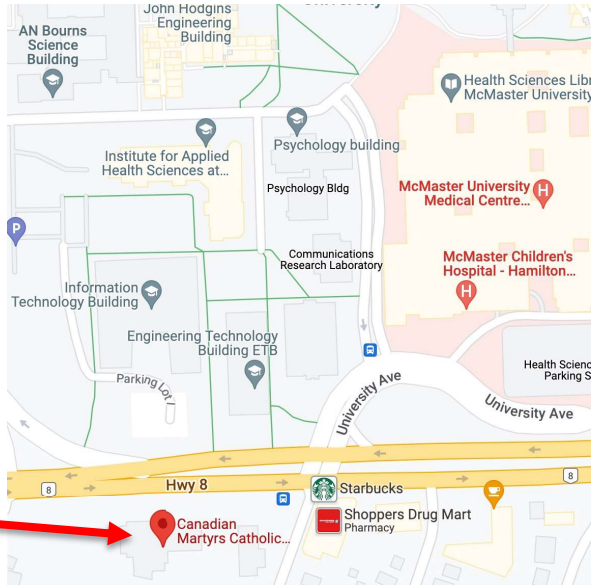
Instructor:

Omid Isfahanialamdari

March 16, 2022

Admin

- Mid-Term 2:
  - Wednesday 23 March 2022
  - Duration: **1 hour**
  - Location: **MCMST CDN_MARTYRS**
    - Seems to be here, I am not sure



  - Covers: Topics from “Doubly Linked Lists” until the lecture of Wednesday 16 March 2022 (inclusive)

Iterating through a Container (Review)

- Let **C** be a **container** and **p** be an **iterator** for **C**

```
for (p = C.begin(); p != C.end(); ++p)
    loop_body
```

- Example: (with an STL vector)
  - Notice how for loop is implemented
  - Notice how we access the element of iterator

```
#include <iostream>
#include <vector>
using namespace std;

int  main(){
    vector<int> v1;
    for (int i = 1; i <= 5; i++)
        v1.push_back(i);

    typedef vector<int>::iterator Iterator;
    int sum = 0;
    for (Iterator p = v1.begin(); p != v1.end(); ++p){
        cout << "Iterator is on data " << *p << endl;
        sum += *p;
    }
    cout << "The sum is :" << sum << endl;

}
```

Output:
Iterator is on data 1
Iterator is on data 2
Iterator is on data 3
Iterator is on data 4
Iterator is on data 5
The sum is :15

Implementing Iterators (remained from pre. lecture)

- Array-based
  - **Array A of the n elements**
  - **index i** that keeps track of the cursor
  - **begin()** = 0
  - **end()** = n (index following the last element)
- Linked list-based
  - For example: A doubly linked-list **L** storing the elements, with sentinels for header and trailer
  - **pointer to node** containing the current element
  - **begin()** = front node (immediately after the header)
  - **end()** = trailer node (immediately after last node)

Implementation of Iterator for (Node) List

- This is an implementation of the iterator for our NodeList (aka DLL)

```
class Iterator {                                  // an iterator for the list
public:
  Elem& operator*();                              // reference to the element
  bool operator==(const Iterator& p) const; // compare positions
  bool operator!=(const Iterator& p) const;
  Iterator& operator++();                         // move to next position
  Iterator& operator--();                         // move to previous position
  friend class NodeList;                          // give NodeList access
private:
  Node* v;                                        // pointer to the node
  Iterator(Node* u);                              // create from node
};
```

```
NodeList::Iterator::Iterator(Node* u)          // constructor from Node*
  { v = u; }

Elem& NodeList::Iterator::operator*()          // reference to the element
  { return v->elem; }
                                               // compare positions
bool NodeList::Iterator::operator==(const Iterator& p) const
  { return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const
  { return v != p.v; }
                                               // move to next position
NodeList::Iterator& NodeList::Iterator::operator++()
  { v = v->next; return *this; }
                                               // move to previous position
NodeList::Iterator& NodeList::Iterator::operator--()
  { v = v->prev; return *this; }
```

```
struct Node {
  Elem elem;
  Node* prev;
  Node* next;
};
```

(Node) List ADT (duplicate slide)

- This implementation is basically a Doubly Linked-List
- **n**:
  - The number of data elements
- Iterator is an implementation of the Position ADT
  - We use iterator instead of a pointer to a specific node in this implementation

```
typedef int Elem;                                  // list base element type
class NodeList {                                   // node-based list
private:
  // insert Node declaration here...
public:
  // insert Iterator declaration here...
public:
  NodeList();                                      // default constructor
  int size() const;                                // list size
  bool empty() const;                              // is the list empty?
  Iterator begin() const;                          // beginning position
  Iterator end() const;                            // (just beyond) last position
  void insertFront(const Elem& e);                 // insert at front
  void insertBack(const Elem& e);                  // insert at rear
  void insert(const Iterator& p, const Elem& e); // insert e before p
  void eraseFront();                               // remove first
  void eraseBack();                                // remove last
  void erase(const Iterator& p);                   // remove p
  // housekeeping functions omitted...
private:                                           // data members
  int     n;                                       // number of items
  Node*  header;                                   // head-of-list sentinel
  Node*  trailer;                                  // tail-of-list sentinel
};
```

STL Iterators in C++

- Each STL container type C supports iterators:
  - **C::iterator** – read/write iterator type
  - **C::const_iterator** – read-only iterator type
  - **C.begin(), C.end()** – return start/end iterators
- This iterator-based operators and methods:
  - ***p:** access current element
  - **++p, --p:** advance to next/previous element
  - **C.assign(p, q):** replace C with contents referenced by the iterator range [p, q) (from p up to, but not including, q)
  - **insert(p, e):** insert e prior to position p
  - **erase(p):** remove element at position p
  - **erase(p, q):** remove elements in the iterator range [p, q)

STL List in C++

```
#include <list>
using std::list;                    // make list accessible
list<float> myList;                 // an empty list of floats
```

list(n): Construct a list with n elements; if no argument list is given, an empty list is created.

size(): Return the number of elements in L.

empty(): Return true if L is empty and false otherwise.

front(): Return a reference to the first element of L.

back(): Return a reference to the last element of L.

push_front(e): Insert a copy of e at the beginning of L.

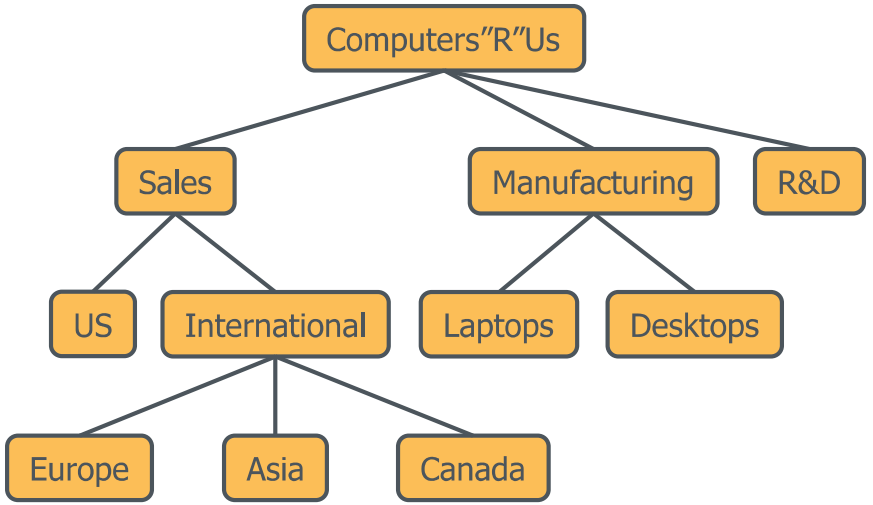
push_back(e): Insert a copy of e at the end of L.

pop_front(): Remove the fist element of L.

pop_back(): Remove the last element of L.

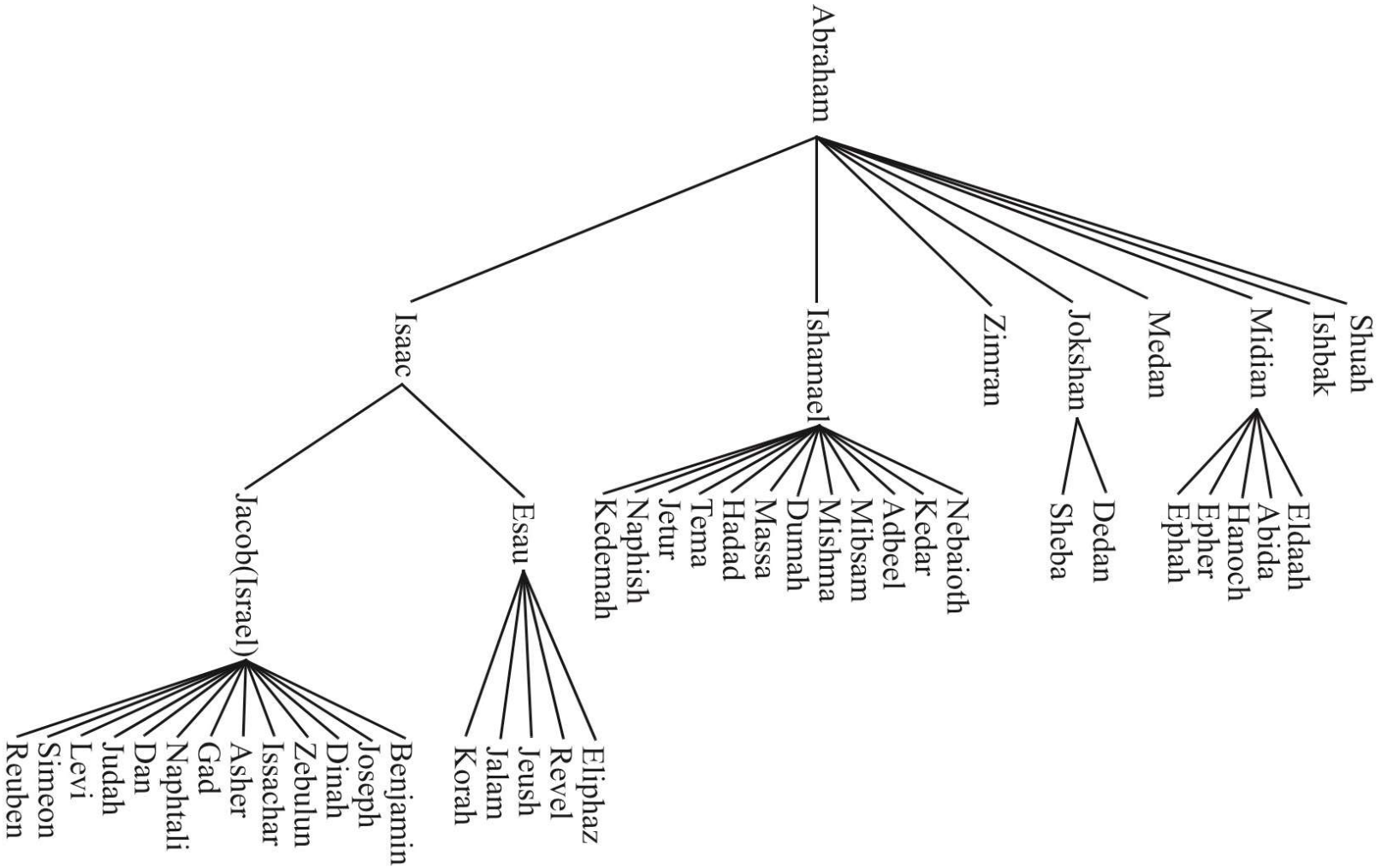
Trees

- So far you are familiar with:
  - Storing elements in a linear fashion
  - Containers and iterators
- In computer science, a tree is an abstract model of a hierarchical structure
- The relation between elements is non-linear
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



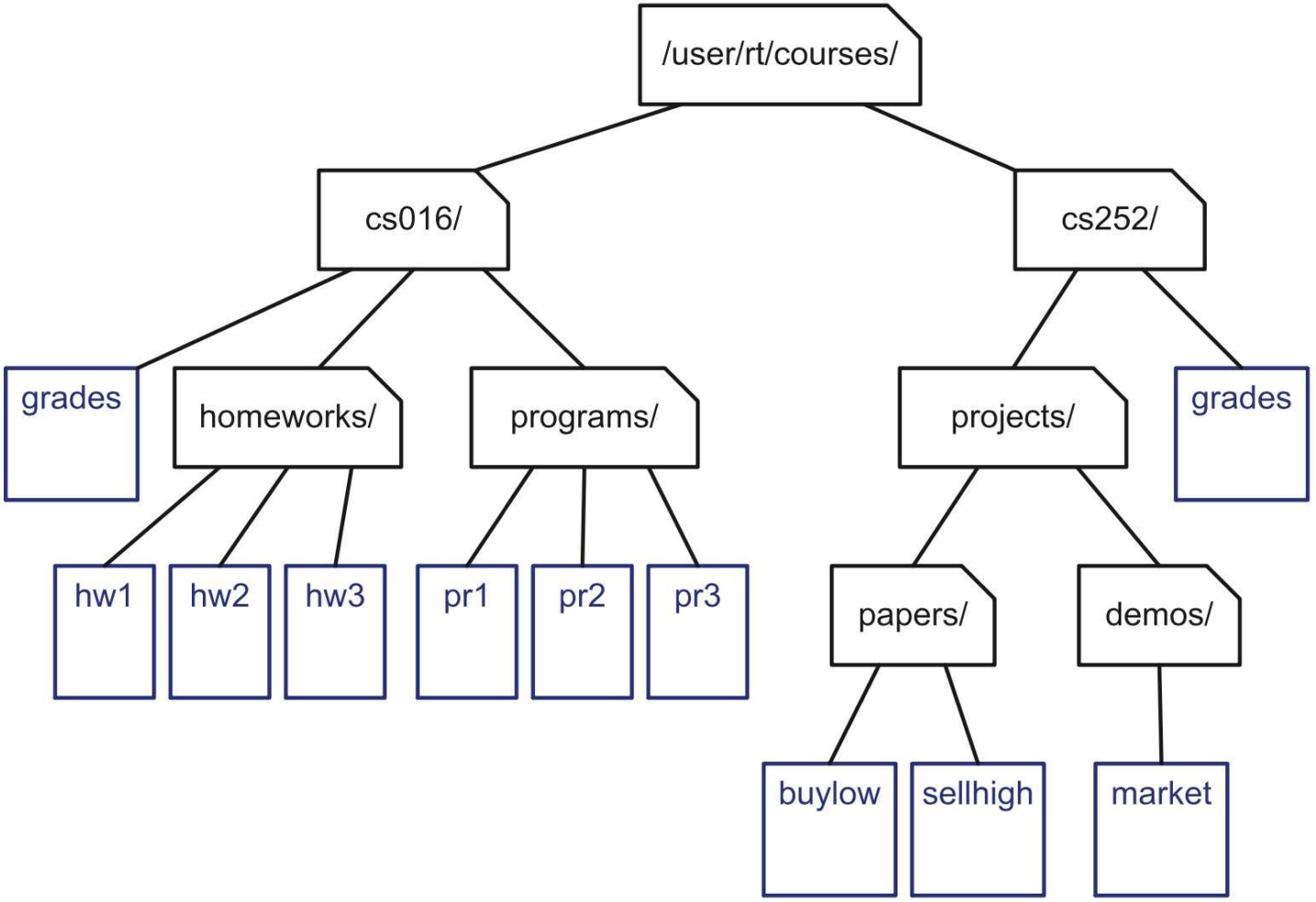
Tree Examples

- Family Tree



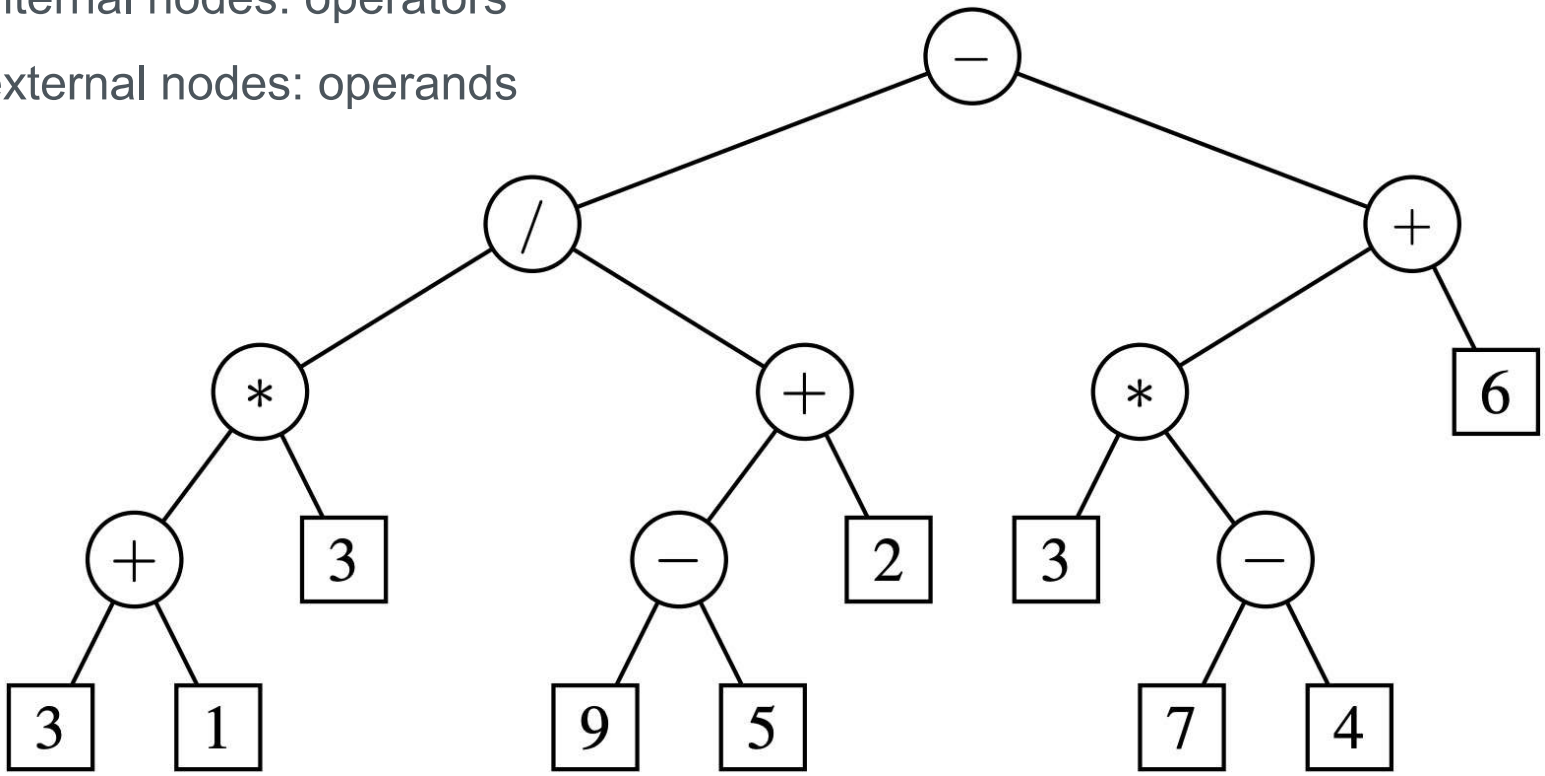
Tree Examples

- File System Tree



Tree Examples

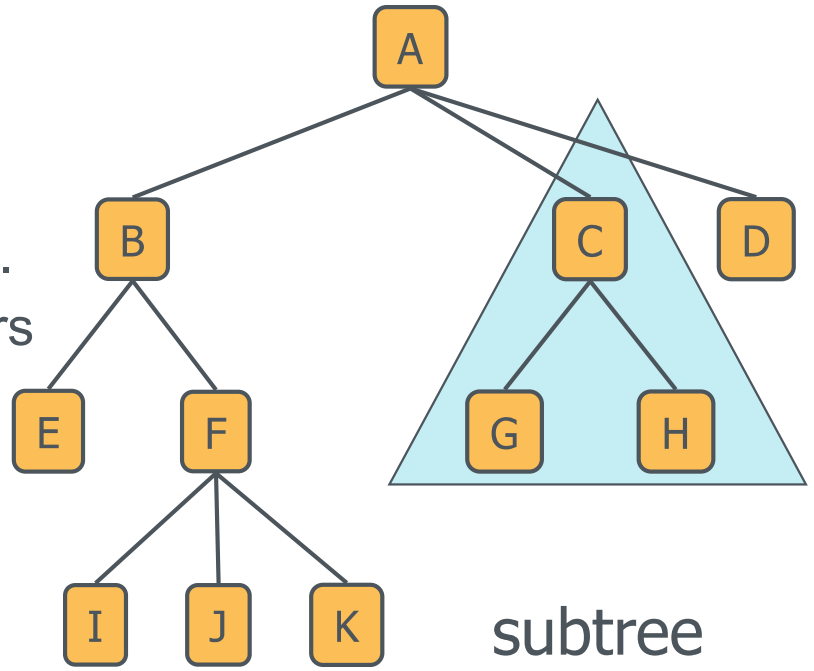
- Expression Tree
  - Binary tree associated with an arithmetic expression
    - internal nodes: operators
    - external nodes: operands



Tree represents: ((((3 + 1) ∗ 3)/((9 − 5) + 2)) − ((3 ∗ (7 − 4)) + 6))

Trees

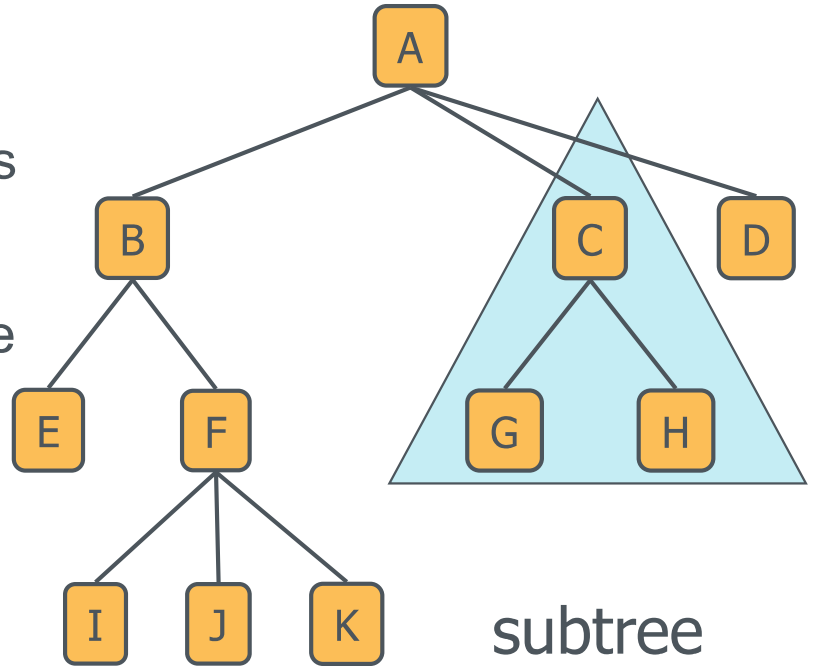
- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.



- Subtree: tree consisting of a node and its descendants

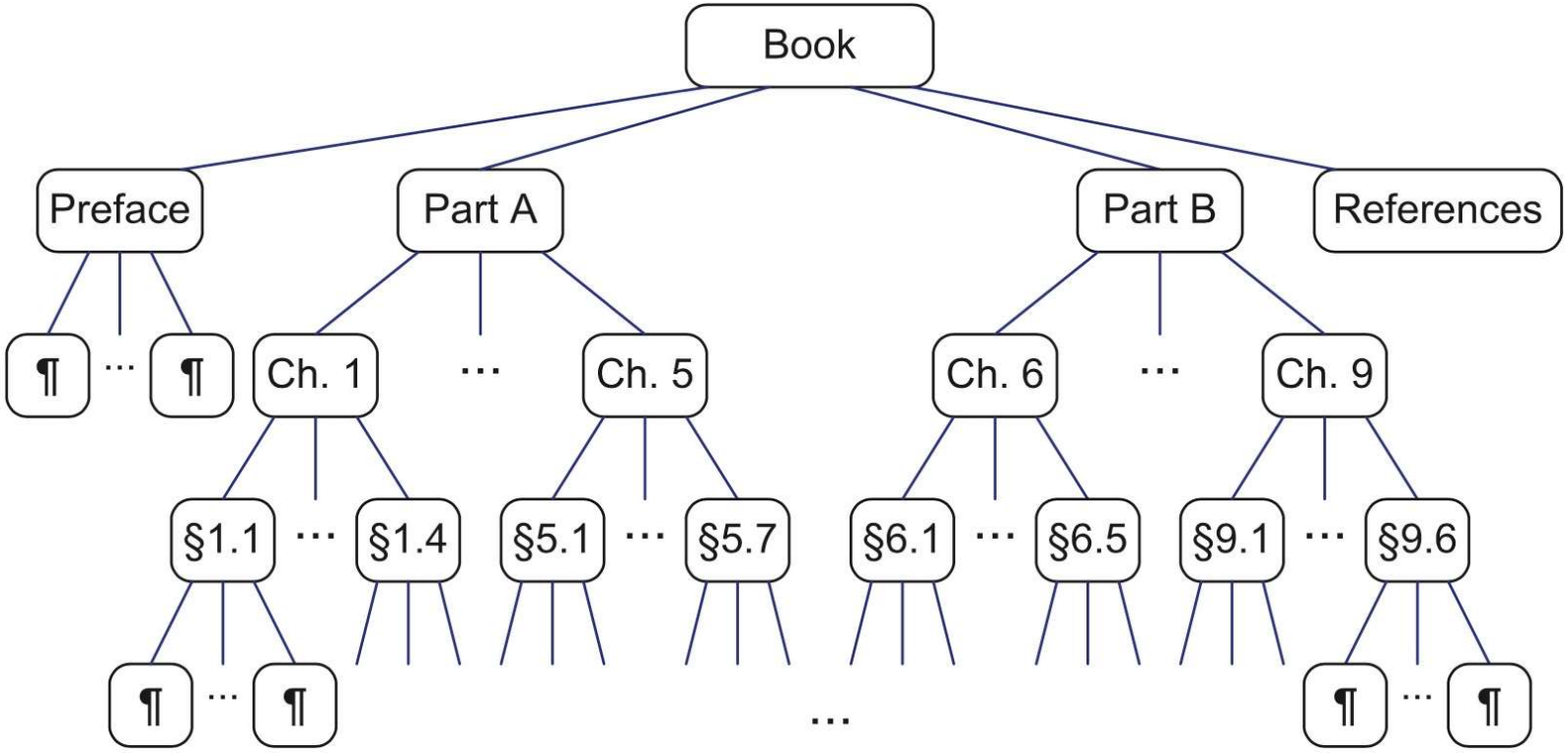
Formal Definition of Tree

- we define tree **T** to be a **set of nodes** storing elements in a parent-child relationship with the following properties:
  - If **T** is *nonempty*, it has a special node, called the **root** of T, that has no parent.
  - Each node *v* of **T** different from the root has a unique **parent** node *w*; every node with parent *w* is a **child** of *w*.
- A tree can be empty!
- So, with the above definition we can define it recursively:
  - a tree **T** is either empty or consists of a node **r**, called the root of **T**, and a (possibly empty) set of trees whose roots are the children of **r**



Ordered Tree Examples

- Ordered Tree: A tree is ordered if there is a linear ordering defined for the children of each node
  - linear order relationship existing between siblings
- Book Organization Tree



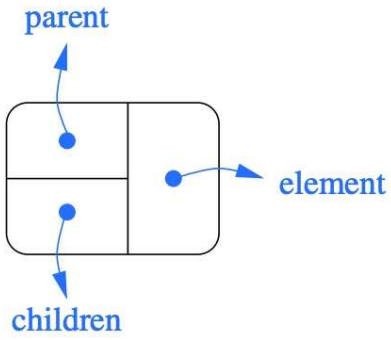
Tree ADT

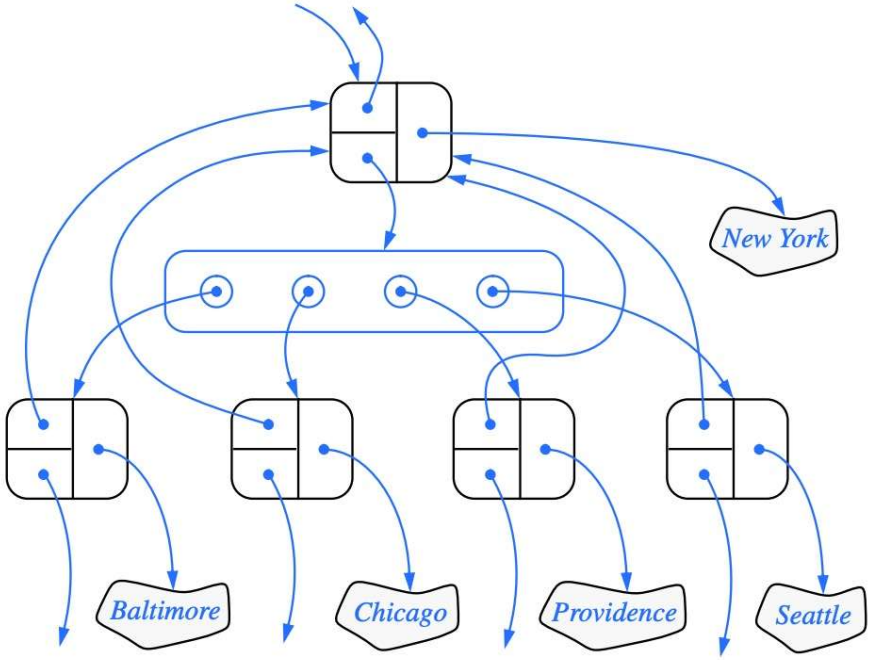
- We use **positions** to abstract nodes
- Generic methods:
  - integer **size**()
  - boolean **empty**()
- Accessor methods:
  - position **root**()
  - list<position> **positions**()
- Position-based methods:
  - position p.**parent**()
  - list<position> p.**children**()
- Query methods:
  - boolean p.**isRoot**()
  - boolean p.**isExternal**()
- Additional update methods may be defined by data structures implementing the Tree ADT

Tree ADT

- General Tree Implementation

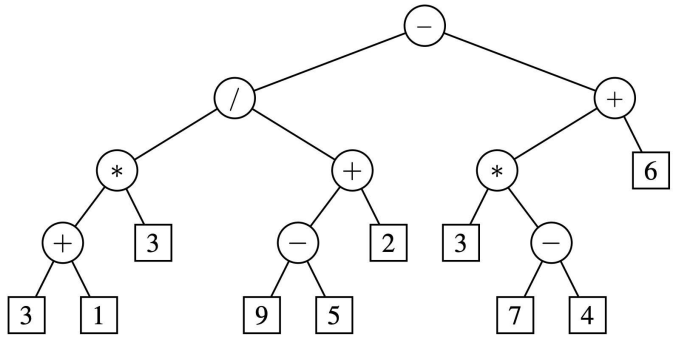
```
template <typename E>                  // base element type
class Position<E> {                    // a node position
public:
  E& operator*();                      // get element
  Position parent() const;             // get parent
  PositionList children() const;       // get node's children
  bool isRoot() const;                 // root node?
  bool isExternal() const;             // external node?
};
```





Tree Traversals

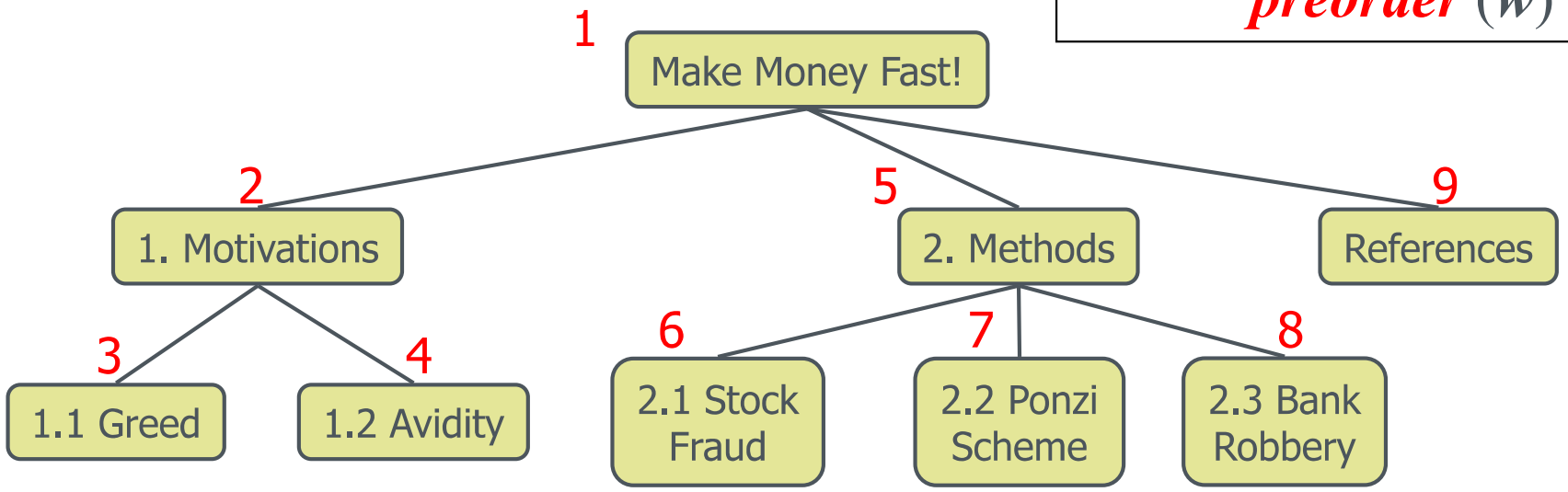
- We were able to easily traverse a linear data structure
  - The traversal of a tree is not trivial
- A **traversal** of a tree **T** is a systematic way of accessing or visiting all the nodes of T.
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal
  - Breadth-first traversal



Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

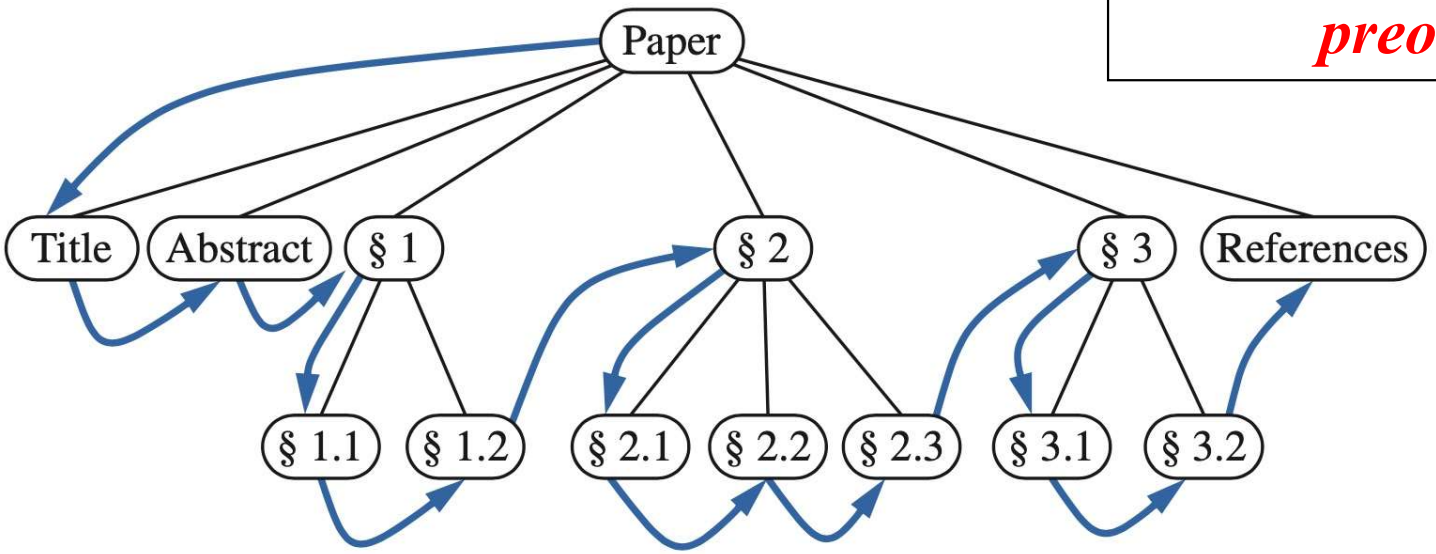
Algorithm ***preOrder(v)***
    visit(v)
    for each child ***w*** of ***v***
        preorder ***(w)***



Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preorder (w)
```



Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

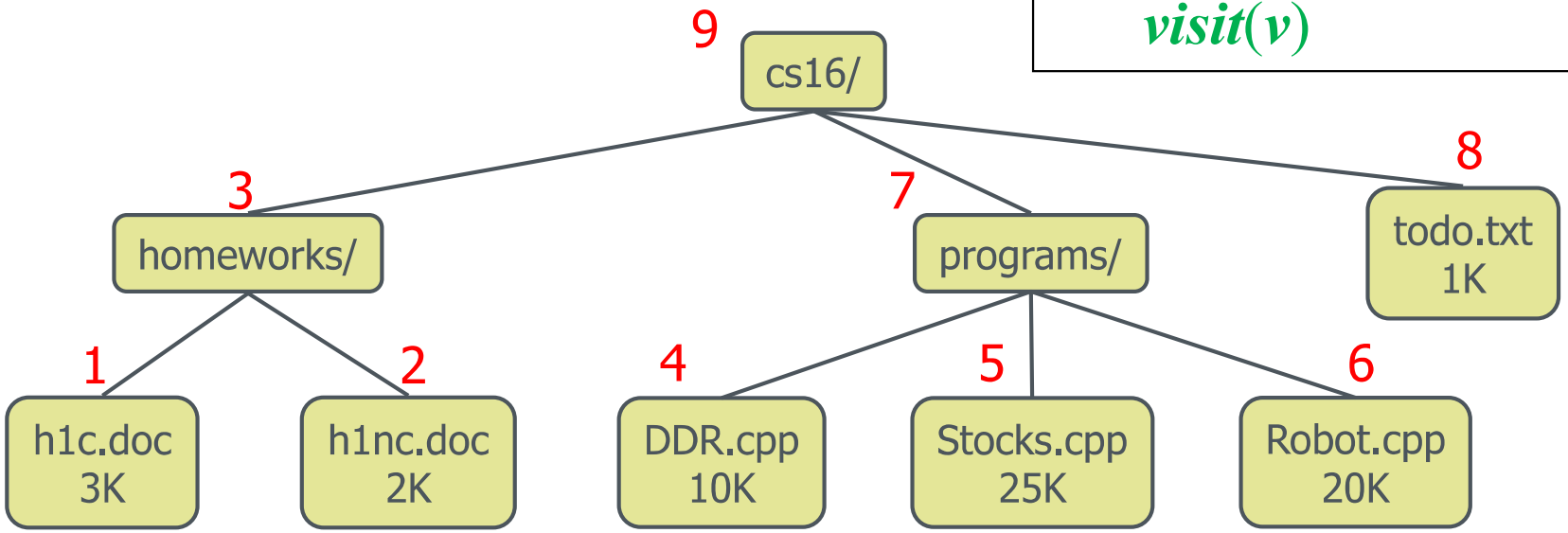
Algorithm ***preOrder(v)***
    visit(v)
    for each child ***w*** of ***v***
        preorder ***(w)***

```
void preorderPrint(const Tree& T, const Position& p) {
  cout << *p;                              // print element
  PositionList ch = p.children();          // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    cout << " ";
    preorderPrint(T, *q);
  }
}
```

Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

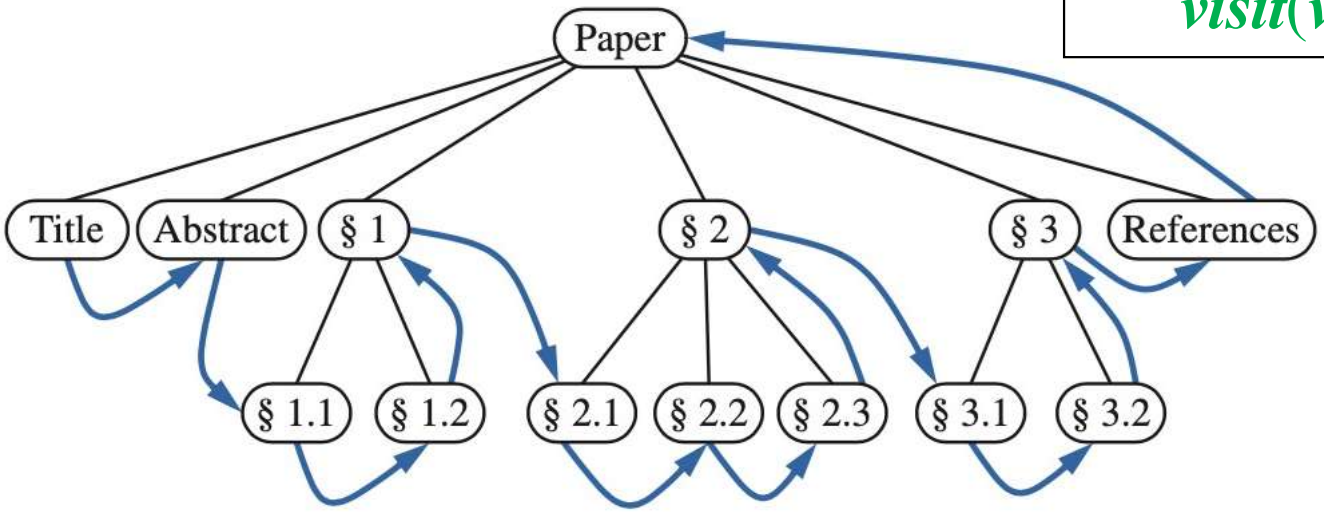
Algorithm ***postOrder(v)***
  for each child ***w*** of ***v***
    postorder ***(w)***
  visit(v)



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

```
Algorithm postOrder(v)
    for each child w of v
        postorder (w)
    visit(v)
```



Postorder Traversal

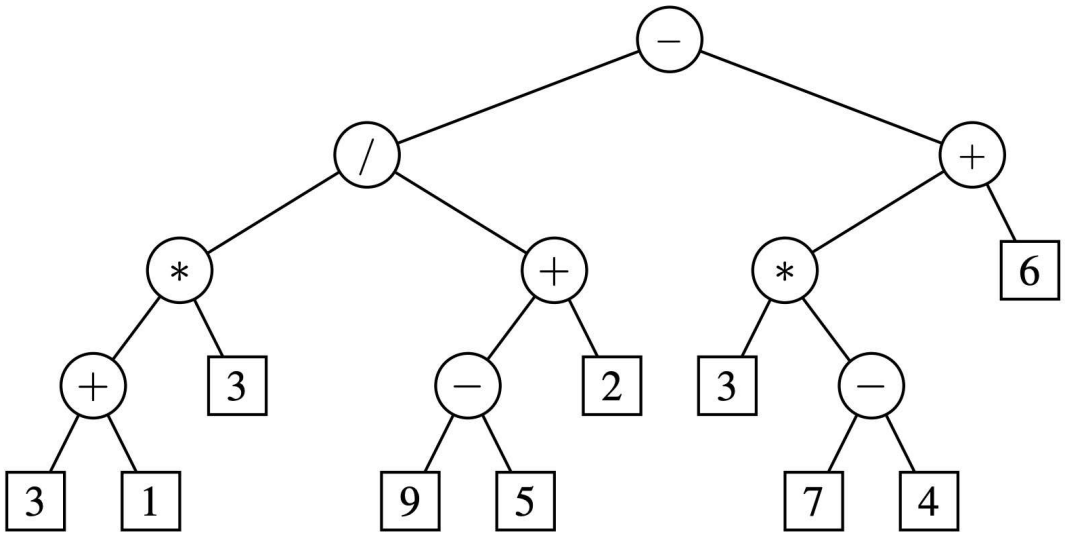
- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm ***postOrder(v)***
  for each child ***w*** of ***v***
    postorder ***(w)***
  visit(v)

```
void postorderPrint(const Tree& T, const Position& p) {
  PositionList ch = p.children();              // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    postorderPrint(T, *q);
    cout << " ";
  }
  cout << *p;                                  // print element
}
```

Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories



Algorithm ***postOrder(v)***
  for each child ***w*** of ***v***
    postorder ***(w)***
  visit(v)

((((3 + 1) ∗ 3)/((9 − 5) + 2)) − ((3 ∗ (7 − 4)) + 6))

Postorder?

Questions?