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

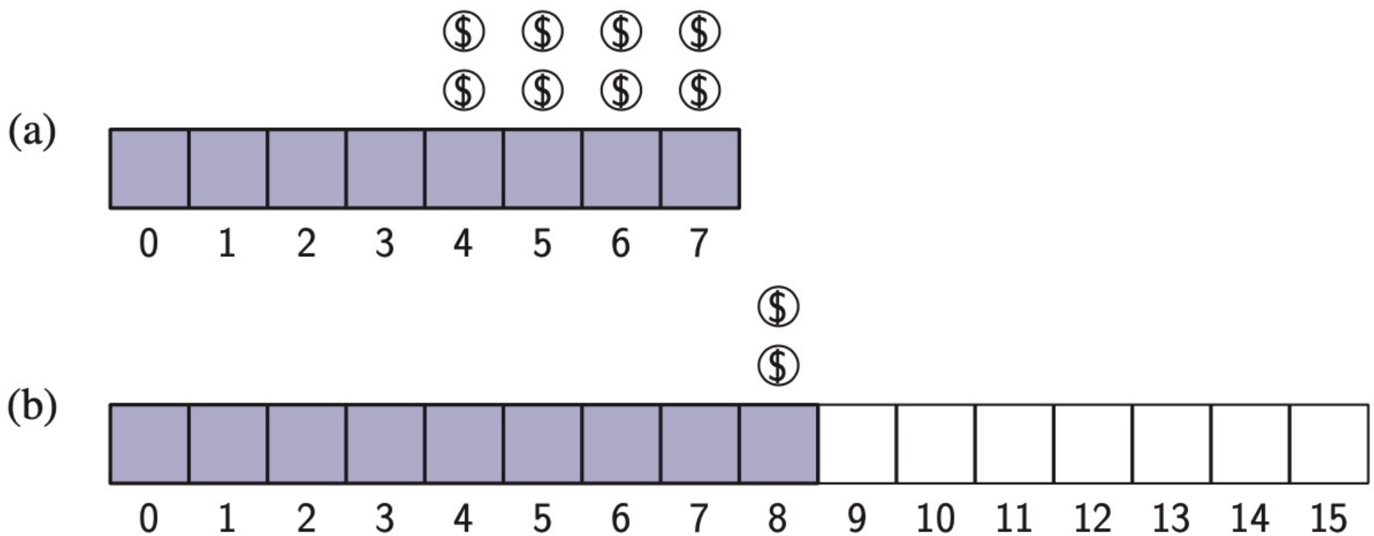
- Suppose we are doing only push operations
- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n insert(o) operations
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling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

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

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

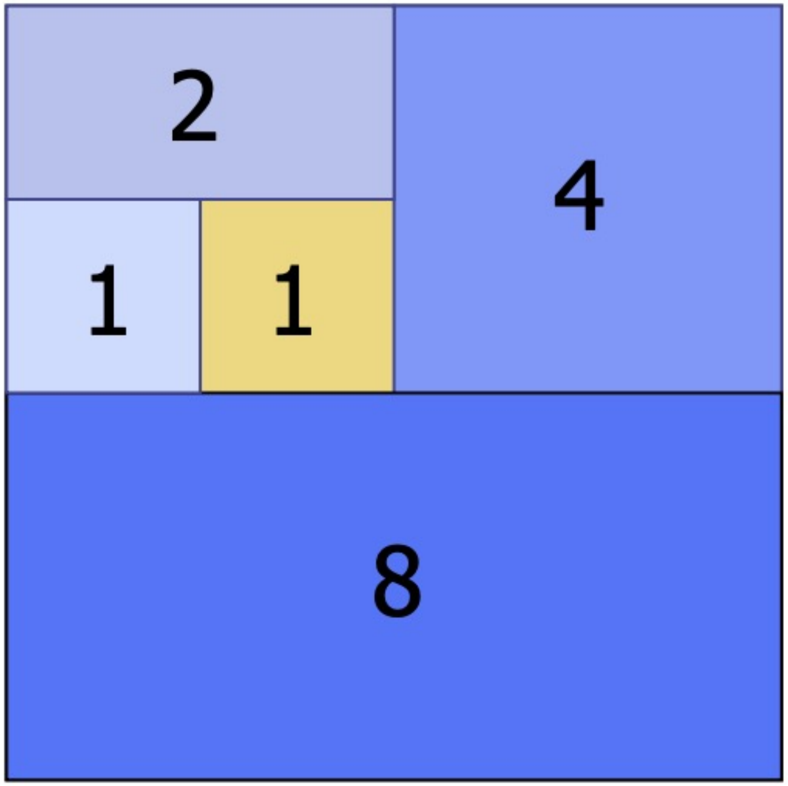
$$n + ck(k + 1)/2$$

- 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

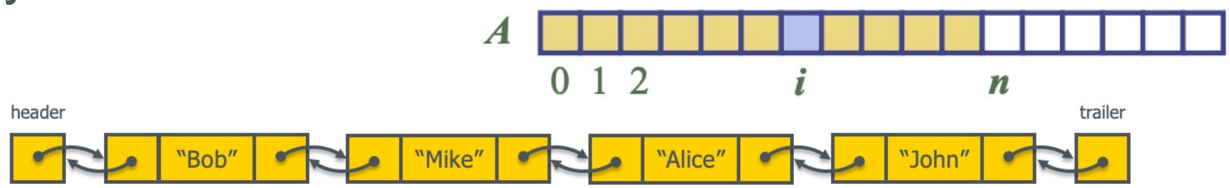
- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n insert operations is proportional to
- $n + 2 + 4 + 8 + \ldots + 2^k =$
  $n + 2^{k+1} - 1 =$
- $3n - 1$
- $T(n)$ is $O(n)$
- The amortized time of an insert operation is $O(1)$

geometric series



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 <u>extends</u> 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

```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;

}
```

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

- 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 <u>header</u> and <u>trailer</u>
  - **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.

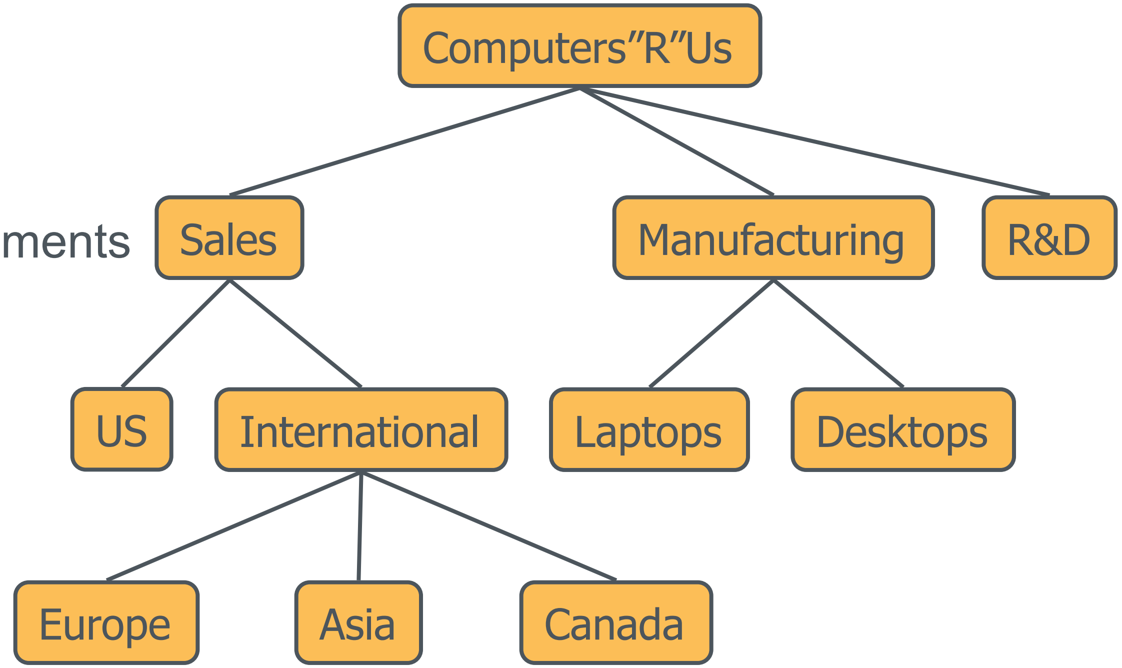
push_back(e): Insert a copy of e at the end of L.

pop_front(): Remove the fist element of L.

pop_back(): Remove the last element of L.

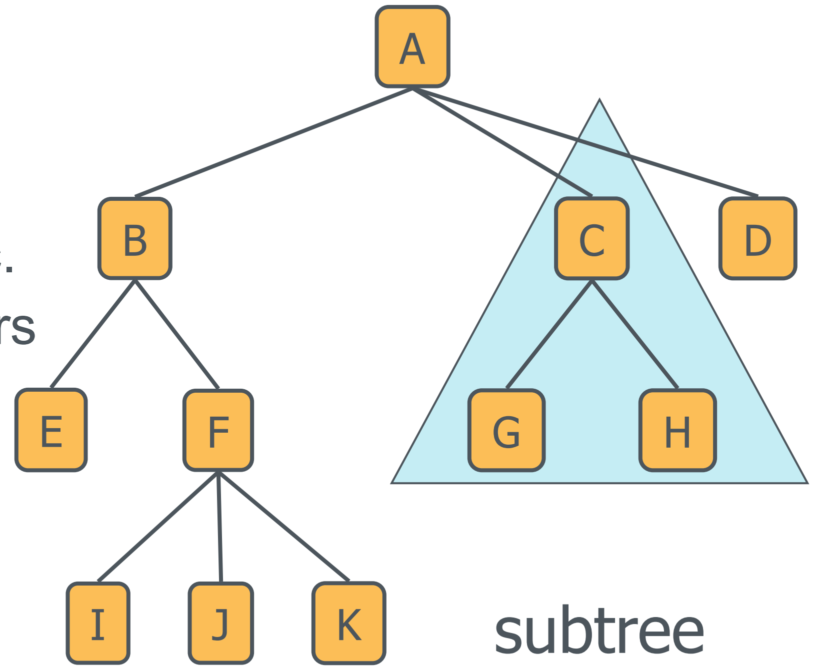
Trees

- 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?