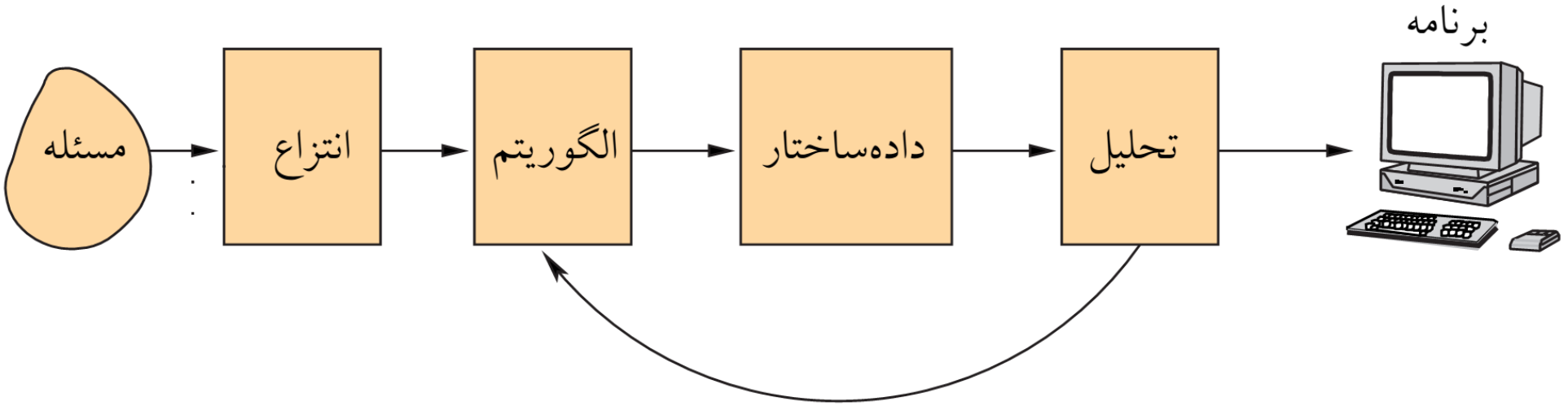
بسم الله الرحمن الرحیم

ساختمان‌های داده

جلسه ۱۱

مجتبی خلیلی
دانشکده برق و کامپیوتر
دانشگاه صنعتی اصفهان

مراحل حل مسئله



ساختمان داده/داده ساختار

- ساختمان داده یا داده ساختار: شیوه‌ای برای ذخیره و سازماندهی داده‌ها به منظور تسهیل در دسترسی و اصلاح/تغییر آنها
  - مثال: آرایه، لیست پیوندی، پشته، صف، درخت

داده گونه انتزاعی (ADT)

- توصیف ریاضی/انتزاعی یک داده به همراه مجموعه عملگرهای آن
  - ورودی و خروجی‌ها را تعریف می‌کند.
  - از جزئیات و پیاده‌سازی حرفی نمیزند.

- پیاده‌سازی:
  - پیاده‌سازی شده یک ADT، ساختمان داده است.

- کاربران، ساختمان داده را به صورت یک ADT می‌بینند.

نخستین ADT: لیست

- لیست: مجموعه‌ای از عناصر با ترتیب مشخص
  - در حالت کلی عناصر میتوانند از هر نوعی باشند.
  - ترتیب یعنی عنصر اول، دوم، ...
  - اندازه آن متغیر است.

عملگرهای لیست

- لیست: مجموعه‌ای از عناصر با ترتیب مشخص
  - برخی از عملگرهای آن عبارتند از:
    - اضافه کردن یک عنصر به ابتدا (addAtfront)
    - اضافه کردن یک عنصر به انتها (addAtback)
    - حذف یک عنصر (delete/remove)
    - جستجوی یک عنصر (search)
    - اندازه لیست (size)
    - چک کردن خالی بودن (empty)
    - دسترسی به یک عنصر (get)

پیاده‌سازی لیست

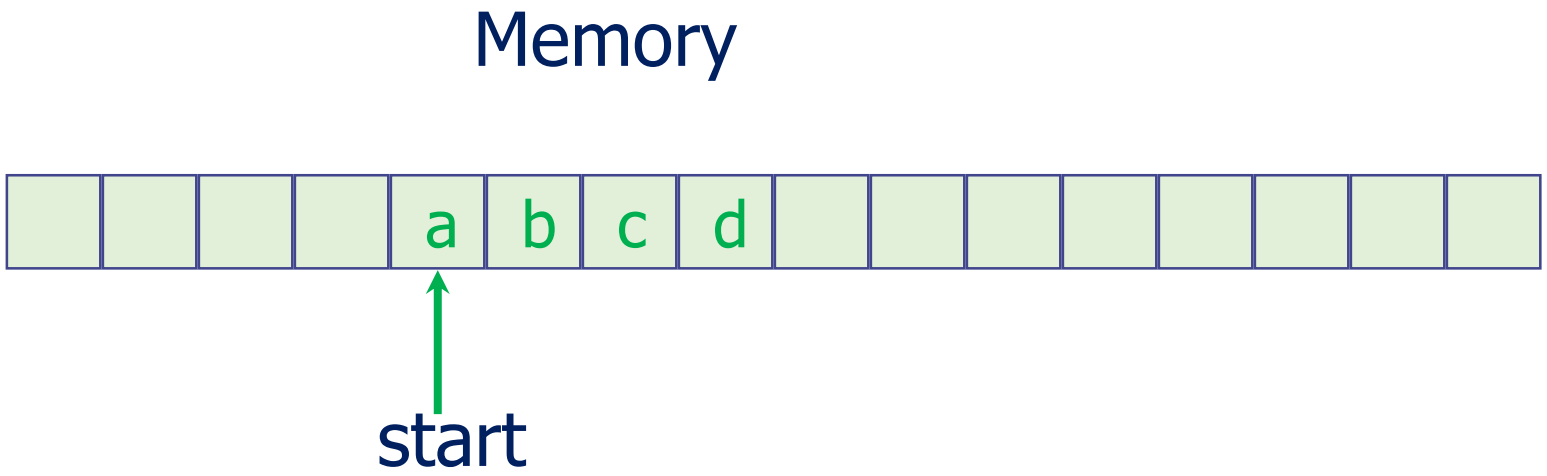
- ADT فقط درباره عملگرها صحبت می‌کند و درباره پیاده‌سازی حرفی نمیزند.

- دو پیاده‌سازی متداول:
  - ArrayList
  - LinkedList

پیاده‌سازی لیست

- درباره کارآمدی

آرایه



- قرار دادن داده‌ها (هم نوع) به صورت پشت سرهم در حافظه
- دسترسی به هر عنصر با استفاده از اندیس
- بسیار ساده و پرکاربرد

آرایه

- آیا میتوان از آرایه c++ به عنوان یک لیست استفاده کرد؟

```
int example()
{
  int arr[20];
  return 0;
}
```

- آرایه در c++ طول ثابت دارد.
- آرایه با طول قابل تغییر؟

آرایه با اندازه قابل تغییر

- Suppose in an insert(o) operation (without an index), we always insert at the end
- When the array is full, we replace the array with a larger one
- 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
```

آرایه با اندازه قابل تغییر

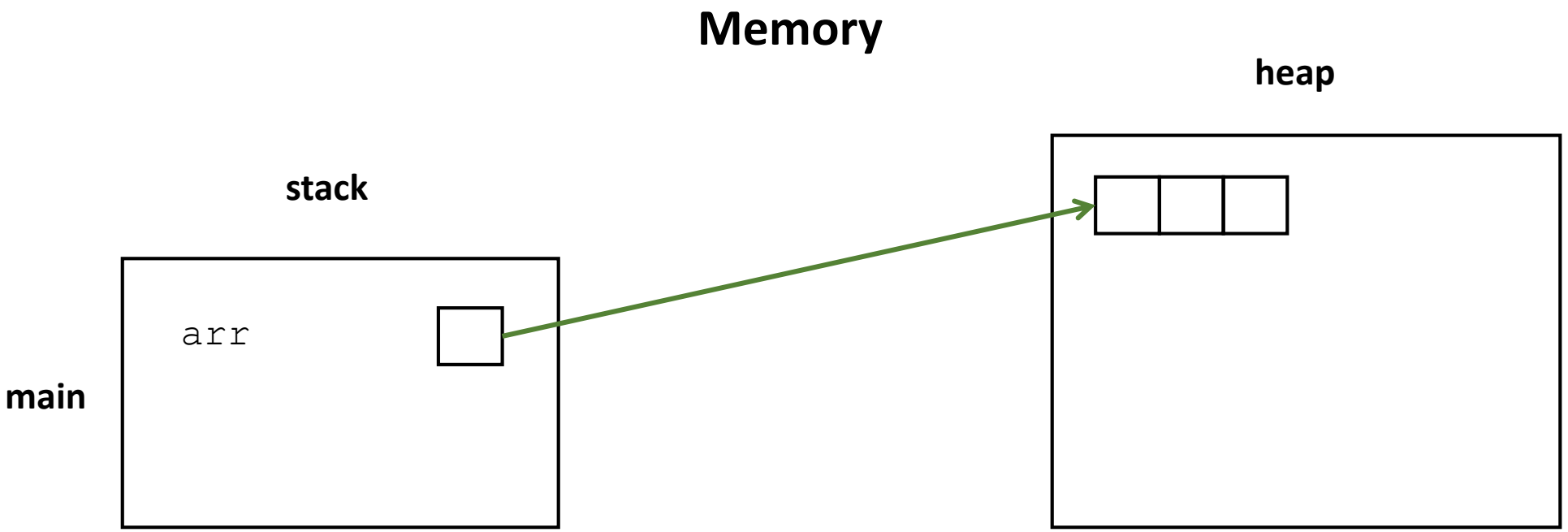
- تخصیص پویای آرایه در heap:

```
int example()
{
  int *arr = new int[3];
  return 0;
}
```

- متغیر آرایه، یک اشاره‌گر به اولین عنصر است.

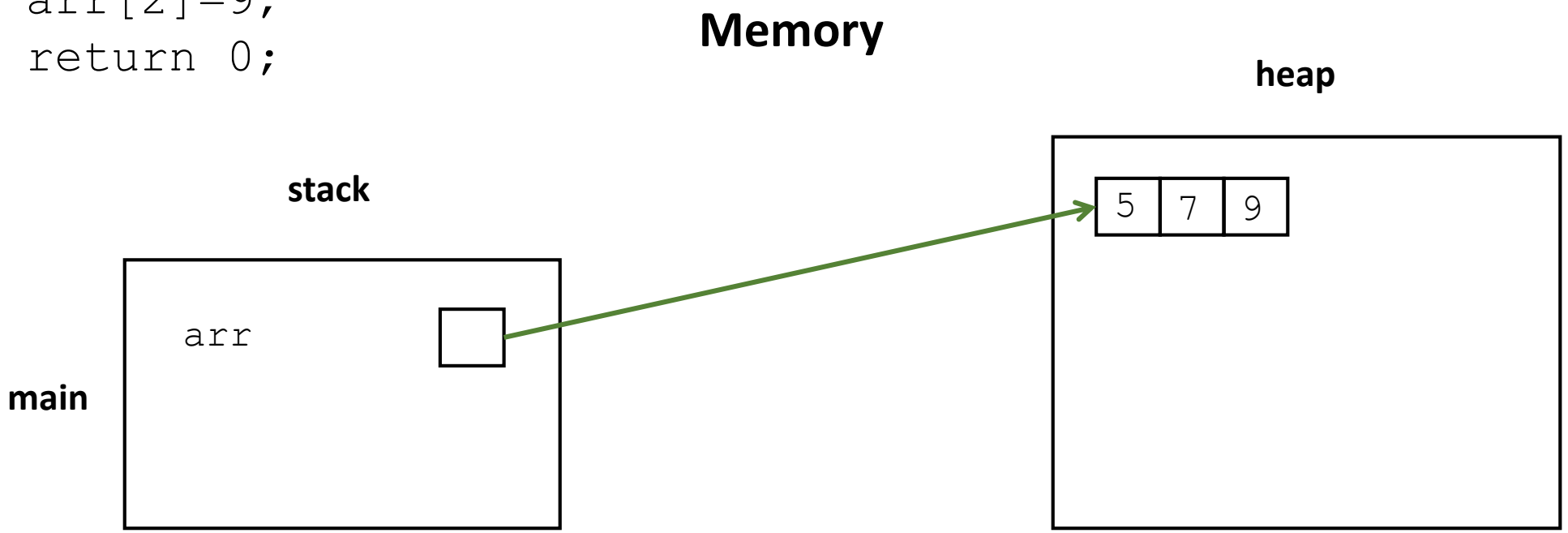
آرایه با اندازه قابل تغییر


```
int example()
{
  int *arr = new int[3];
  return 0;
}
```



آرایه با اندازه قابل تغییر


```
int example()
{
   int *arr = new int[3];
   arr[0]=5;
   arr[1]=7;
   arr[2]=9;
   return 0;
}
```

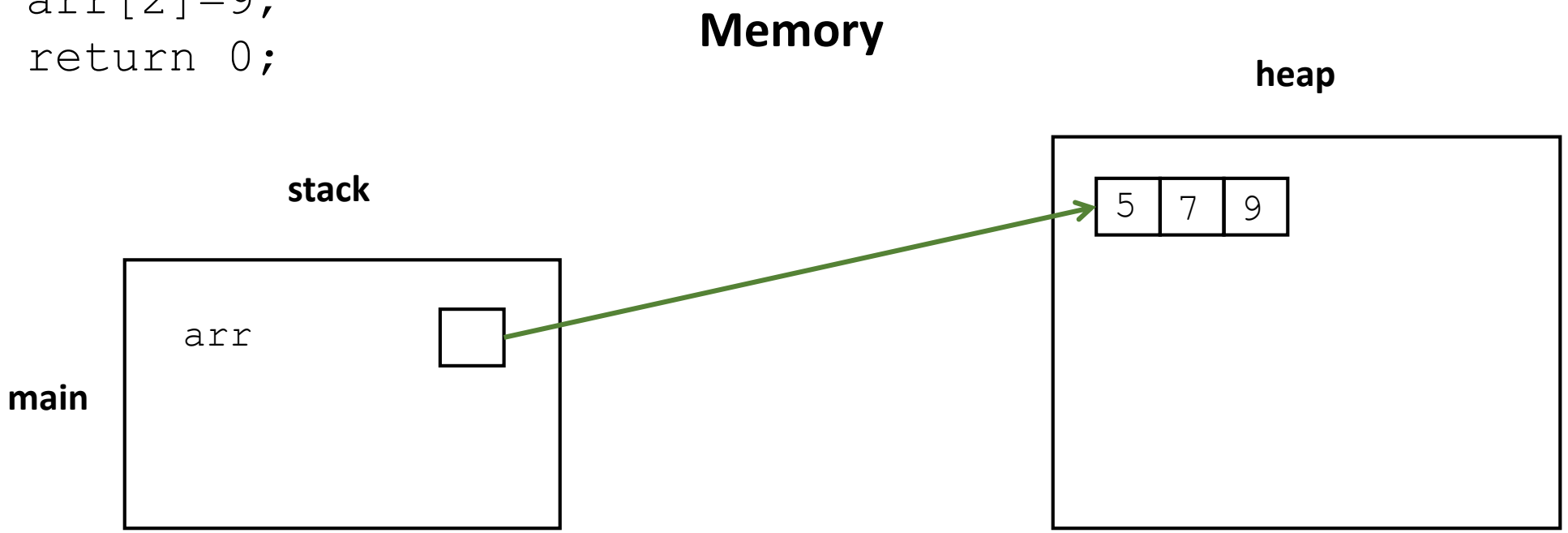


آرایه با اندازه قابل تغییر

IUT-ECE

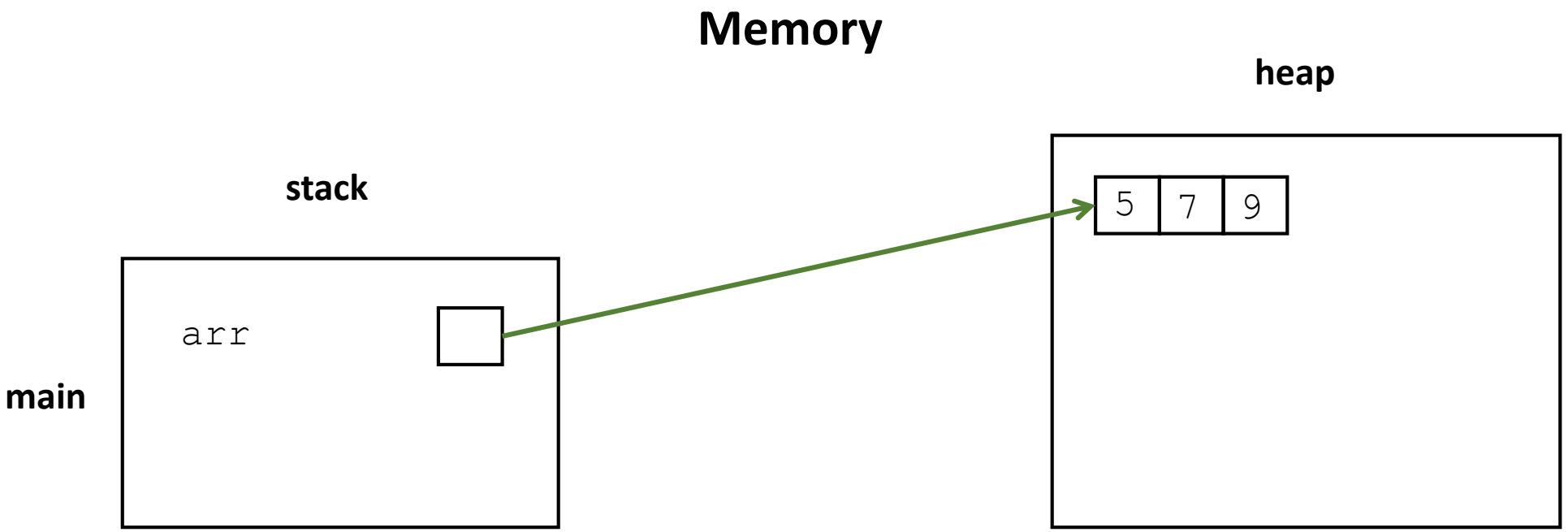
```
int example()
{
   int *arr = new int[3];
   arr[0]=5;
   arr[1]=7;
   arr[2]=9;
   return 0;
}
```

- اگر آرایه پر شد اما نیاز به اندازه بیشتری بود؟

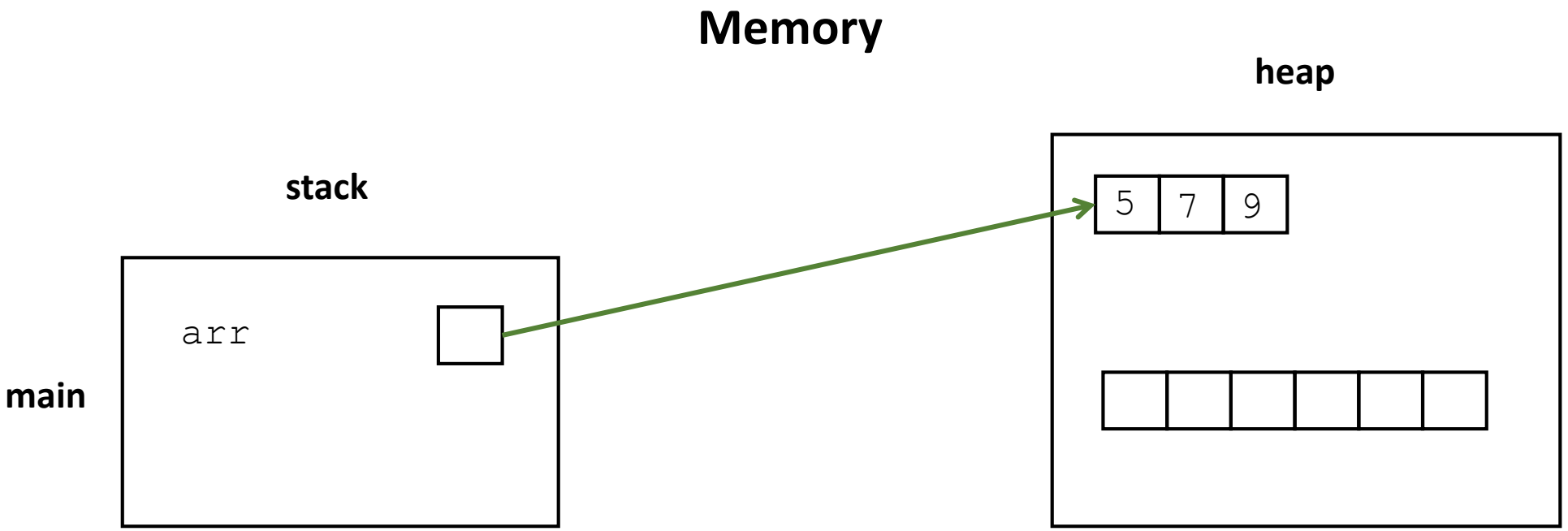


آرایه با اندازه قابل تغییر

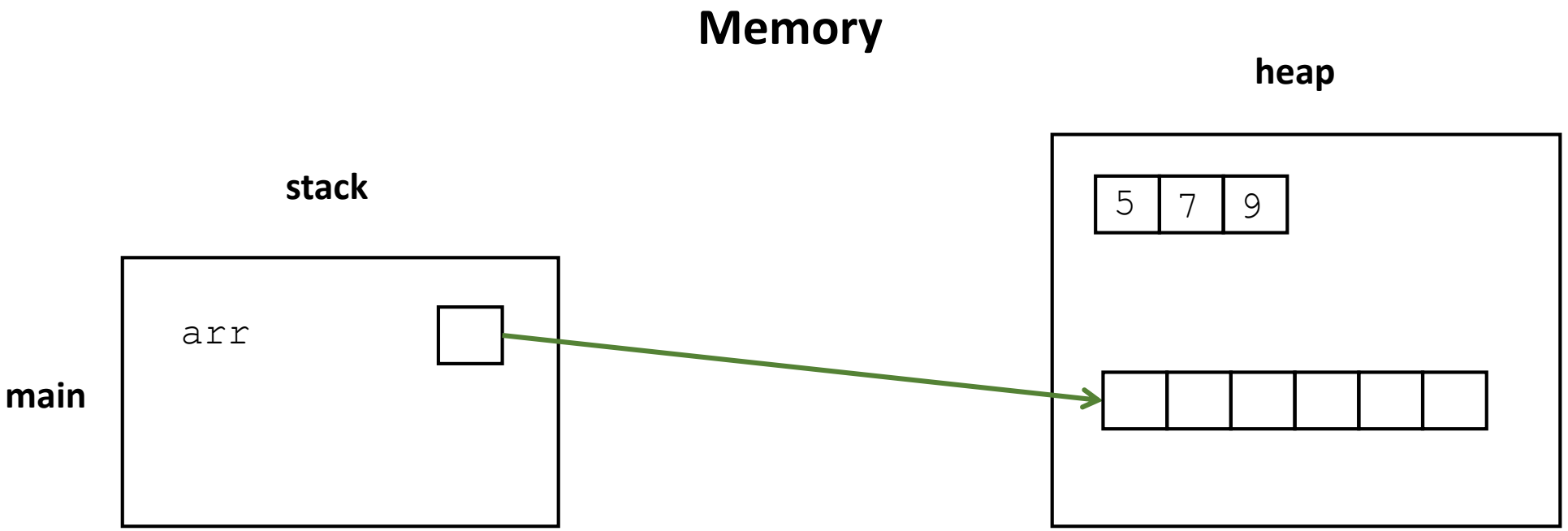
- ایجاد یک آرایه بزرگتر و تخصیص آدرس جدید به arr



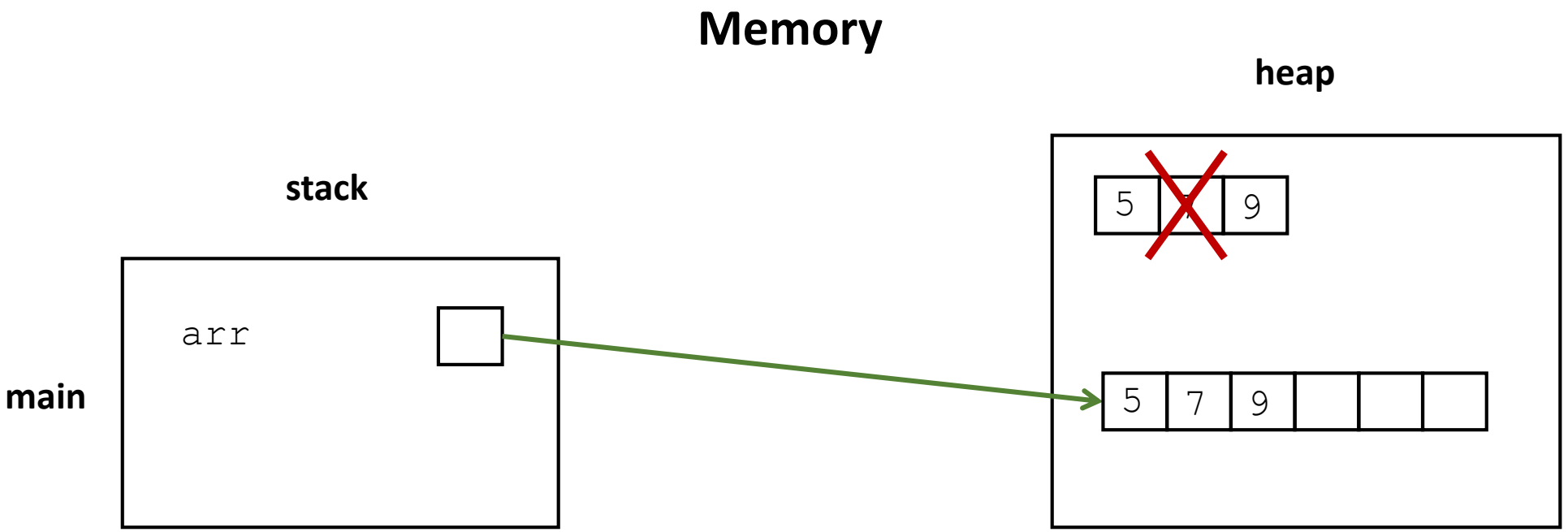
آرایه با اندازه قابل تغییر



آرایه با اندازه قابل تغییر



آرایه با اندازه قابل تغییر



تحلیل

- هر بار اندازه آرایه را یکی اضافه کنیم:
- اگر آخرین اندازه برابر n باشد:

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2} = O(n^2)$$

amortized time per insert operation = O(*n*)

تحلیل

- هر بار اندازه آرایه را دو برابر کنیم:

$n = 1$

$n = 2$

$n = 3$

$n = 4$

$n = 5$

...

$n = 8$

تحلیل

- هر بار اندازه آرایه را دو برابر کنیم:
- اگر آخرین اندازه برابر $2^k < n \leq 2^{k+1}$ باشد:

$$\underbrace{(1+1+1+\ldots+1)}_{\text{هزینه درج}} + \underbrace{(1+2+4+\ldots+2^k)}_{\text{هزینه کپی}} = n + \underbrace{(2^{k+1}-1)}_{\leq 2n} = O(n)$$

$$\underbrace{\qquad\qquad}_{\leq 3n}$$

amortized time per insert operation = O(1)

ArrayList

- The Vector or Array List extends the notion of array
- 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:
  - at(integer i): returns the element at index i without removing it
  - set(integer i, object o): replace the element at index i with o
  - insert(integer i, object o): insert a new element o to have index i
  - erase(integer i): removes element at index i
- Additional methods:
  - size()
  - empty()

ArrayList

IUT-ECE

```
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
};
```

ArrayList

IUT-ECE

```
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];
}
```

ArrayList

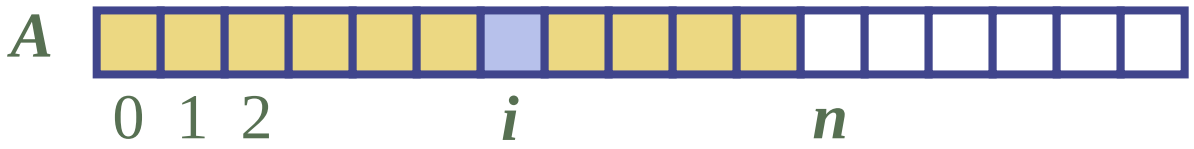
IUT-ECE

```
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
}
```

```
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
}
```

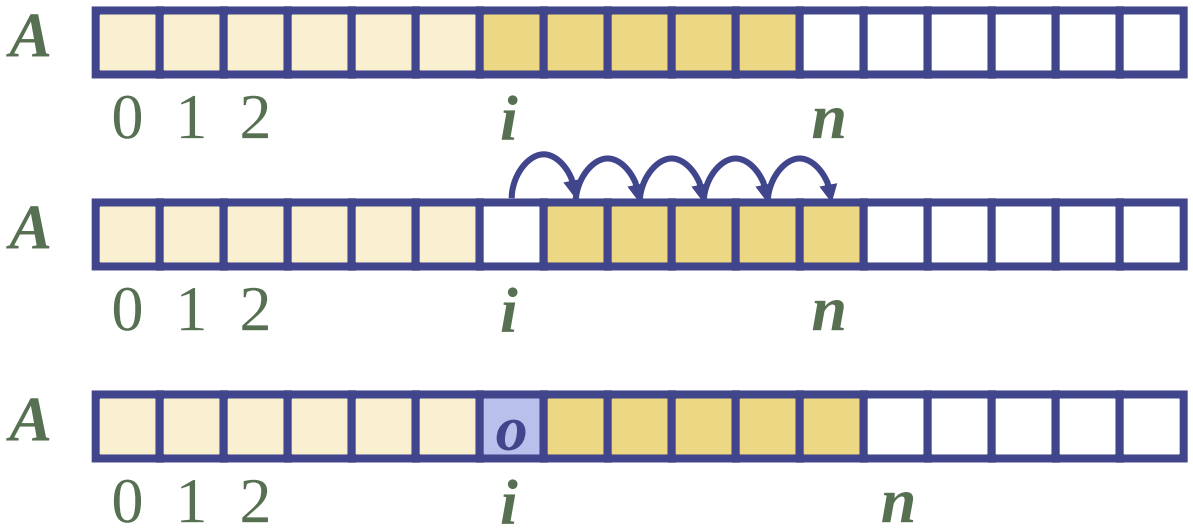
ArrayList

- 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 $A[i] = o$



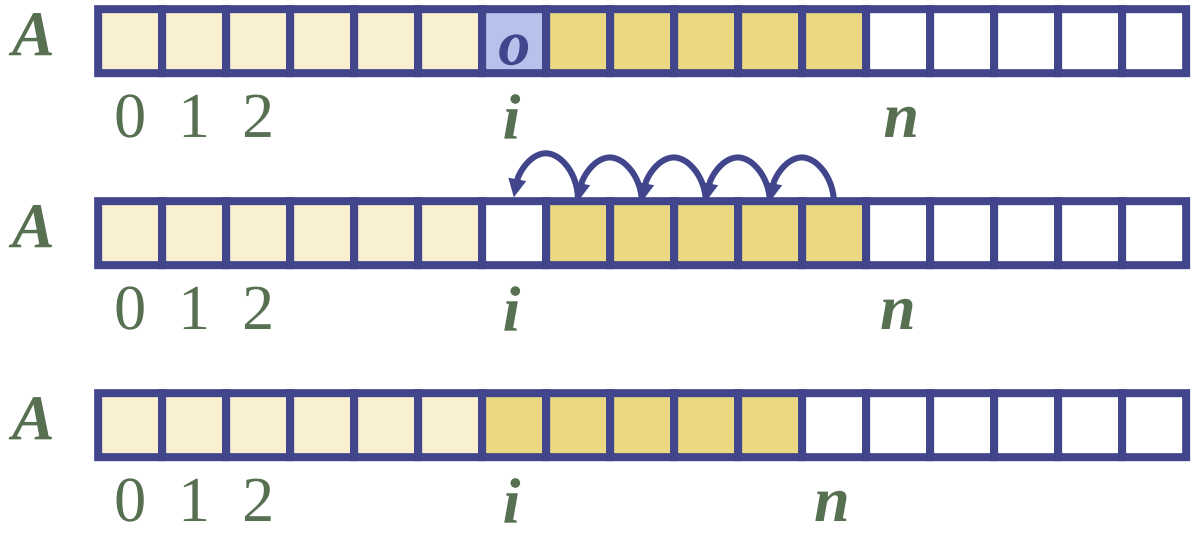
ArrayList

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



ArrayList

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



Vectors

- آرایه‌های قابل تغییر در برخی زبانها به صورت built-in وجود دارند.
  - List در پایتون
- C++ نیز پیاده‌سازی خودش را دارد.

Vectors

- Vectors شبیه آرایه‌ها هستند با این تفاوت که اندازه‌شان قابل تغییر است.

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector<string> names; // no size required.
  names.push_back(“aa"); // adds element
  names.push_back("bb"); // adds element
  names.push_back("cc"); //adds element
```

```
#include <vector>                    // provides definition of vector
using std::vector;                   // make vector accessible

vector<int> myVector(100);           // a vector with 100 integers
```

vector(n): Construct a vector with space for n elements; if no argument is given, create an empty vector.

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

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

resize(n): Resize V, so that it has space for n elements.

reserve(n): Request that the allocated storage space be large enough to hold n elements.

operator[i]: Return a reference to the ith element of V.

at(i): Same as $V[i]$, but throw an out_of_range exception if i is out of bounds, that is, if $i < 0$ or $i \geq V$.size().

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

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

push_back(e): Append a copy of the element e to the end of V, thus increasing its size by one.

pop_back(): Remove the last element of V, thus reducing its size by one.

آرایه

- دسترسی سریع به هر عنصر در آرایه
  - حسن است.

آرایه

- درج یا حذف در آرایه سخت است.
  - به شیفت نیاز دارد تا داده جدیدی درج شود.
  - بعد از حذف نیاز است جای خالی پر شود.

آرایه

- یک بده بستان بین زمان و فضا
  - مثلا دیدید که زمانی که به ظرفیت آرایه رسیدیم اندازه را دو برابر کردیم.
  - هرچند که کپی کردن داده‌ها زمانبر است اما این افزایش اندازه ندرتا ممکن است انجام شود.
  - اما دو برابر کردن فضا می‌تواند باعث هدر دادن فضای حافظه شود. در نظر بگیرید که ممکن است اندازه اولیه داده بسیار بزرگ باشد.

آرایه

- یک بده بستان بین زمان و فضا
  - اگر فقط اندازه را به میزان یک واحد اضافه کنیم.
  - نیاز به دفعات زیاد کپی کردن که از لحاظ زمانی ناکارآمد است.
  - از لحاظ فضا کارآمد است و هدر رفت کمی دارد.

آرایه

- درج یا حذف در آرایه سخت است.
  - به شیفت نیاز دارد تا داده جدیدی درج شود.
  - بعد از حذف نیاز است جای خالی پر شود.

- آیا میتوانیم داده‌ها را به صورت منطقی (بجای فیزیکی) به هم متصل کنیم؟
  - **لیست پیوندی**

Example: Storing Game Entries in an Array

The first application we study is for storing entries in an array; in particular, high score entries for a video game. Storing objects in arrays is a common use for arrays, and we could just as easily have chosen to store records for patients in a hospital or the names of players on a football team. Nevertheless, let us focus on storing high score entries, which is a simple application that is already rich enough to present some important data structuring concepts.

Example: Storing Game Entries in an Array

- Let us focus on storing high score entries.

Mike	Rob	Paul	Anna	Rose	Jack				
1105	750	720	660	590	510				
0	1	2	3	4	5	6	7	8	9

Example: Storing Game Entries in an Array

- A C++ class representing a game entry:

```
class GameEntry {                                   // a game score entry
public:
  GameEntry(const string& n="", int s=0); // constructor
  string getName() const;                           // get player name
  int getScore() const;                             // get score
private:
  string name;                                      // player's name
  int score;                                        // player's score
};
```

Example: Storing Game Entries in an Array

- A C++ class representing a game entry:

```
GameEntry::GameEntry(const string& n, int s) // constructor
  : name(n), score(s) { }

                                              // accessors
string GameEntry::getName() const { return name; }
int GameEntry::getScore() const { return score; }
```

Example: Storing Game Entries in an Array

- A C++ class for storing high game scores:

```
class Scores {                                  // stores game high scores
public:
  Scores(int maxEnt = 10);                      // constructor
  ~Scores();                                    // destructor
  void add(const GameEntry& e);                 // add a game entry
  GameEntry remove(int i)                       // remove the ith entry
      throw(IndexOutOfBounds);
private:
  int maxEntries;                               // maximum number of entries
  int numEntries;                               // actual number of entries
  GameEntry* entries;                           // array of game entries
};
```

Example: Storing Game Entries in an Array

- A C++ class for storing high game scores:

```
Scores::Scores(int maxEnt) {                    // constructor
  maxEntries = maxEnt;                          // save the max size
  entries = new GameEntry[maxEntries];          // allocate array storage
  numEntries = 0;                               // initially no elements
}

Scores::~Scores() {                             // destructor
  delete[] entries;
}
```

Example: Storing Game Entries in an Array

- Till now:

Mike	Rob	Paul	Anna	Rose	Jack				
1105	750	720	660	590	510				
0	1	2	3	4	5	6	7	8	9

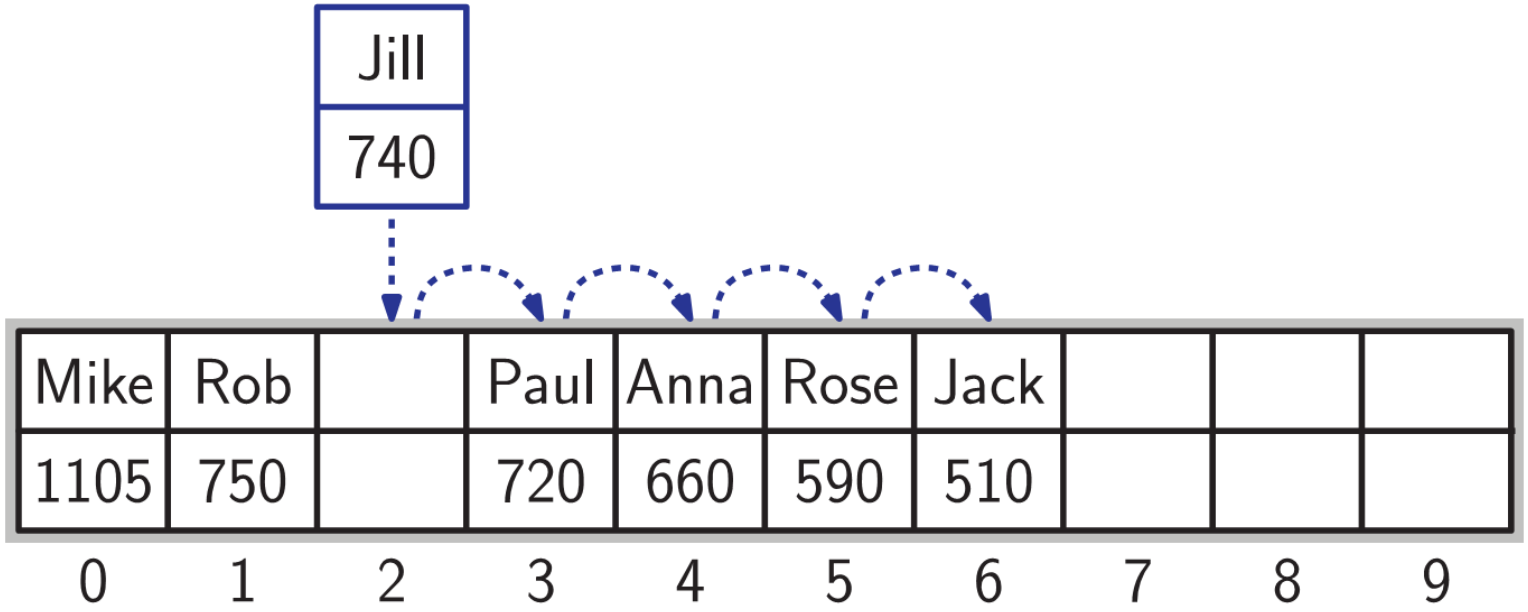
Example: Storing Game Entries in an Array

- Insertion:

add(e): Insert game entry e into the collection of high scores. If this causes the number of entries to exceed *maxEntries*, the smallest is removed.

Example: Storing Game Entries in an Array

- Insertion:



Example: Storing Game Entries in an Array

- Insertion:

```
void Scores::add(const GameEntry& e) {    // add a game entry
  int newScore = e.getScore();            // score to add
  if (numEntries == maxEntries) {         // the array is full
    if (newScore <= entries[maxEntries-1].getScore())
      return;                             // not high enough - ignore
  }
  else numEntries++;                      // if not full, one more entry

  int i = numEntries-2;                   // start with the next to last
  while ( i >= 0 && newScore > entries[i].getScore() ) {
    entries[i+1] = entries[i];            // shift right if smaller
    i--;
  }
  entries[i+1] = e;                       // put e in the empty spot
}
```

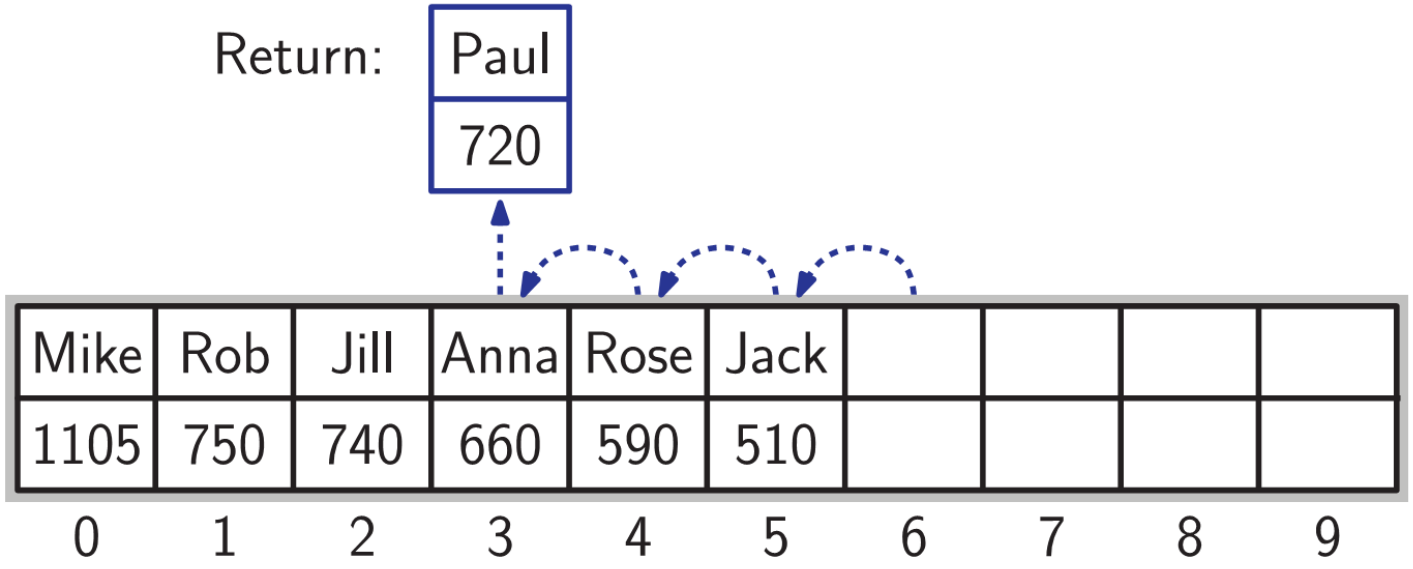
Example: Storing Game Entries in an Array

- Remove:

remove(i): Remove and return the game entry e at index i in the *entries* array. If index i is outside the bounds of the *entries* array, then this function throws an exception; otherwise, the *entries* array is updated to remove the object at index i and all objects previously stored at indices higher than i are "shifted left" to fill in for the removed object.

Example: Storing Game Entries in an Array

- Remove:



Example: Storing Game Entries in an Array

- Remove:

```
GameEntry Scores::remove(int i) throw(IndexOutOfBounds) {
  if ((i < 0) || (i >= numEntries))          // invalid index
    throw IndexOutOfBounds("Invalid index");
  GameEntry e = entries[i];                  // save the removed object
  for (int j = i+1; j < numEntries; j++)
    entries[j-1] = entries[j];               // shift entries left
  numEntries--;                              // one fewer entry
  return e;                                  // return the removed object
}
```

آرایه‌های دو بعدی


```
int M[8][10];          // matrix with 8 rows and 10 columns
```

This statement creates a two-dimensional "array of arrays," M, which is 8×10, having 8 rows and 10 columns. That is, M is an array of length 8 such that each element of M is an array of length 10 of integers. (See Figure 3.6.)

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

آرایه‌های دو بعدی

int A[4][3];

A00	A01	A02	A10	A11	A12	A20	A21	A22	A30	A31	A32
A==A[0]			A[1]			A[2]			A[3]		

address(A[i][j]) = address(A[0][0]) + (i × n + j) × size(int)

آرایه‌های دو بعدی

- با استفاده از vector:

```
vector< vector<int> > M(n, vector<int>(m));
cout << M[i][j] << endl;
```