Data Structures & Algorithms in C++

WEEK 03

Chap 3. Arrays, Linked Lists & Recursion

- Arrays
- Singly Linked Lists
- Doubly Linked Lists
- Circularly Linked Lists
- Recursion

Arrays
- Static Array

- Array is a collection of elements stored in contiguous memory locations
  - All elements must be of the same data type

- What is a Static Array ?
  - Size is fixed during declaration
  - Memory allocated at compile time
  - Fast, easy to use, efficient for small datasets,
    but, waste memory if size is overestimated, cannot resize

```
Int staticArray[10];  // size is fixed at 10
```

Arrays
- Dynamic Array

- What is a Dynamic Array?
  - Size can change at runtime
  - Memory allocated on the heap using pointers
  - Flexible, no wastage of memory,

    but, slower than static arrays, prone to memory leaks if not managed properly

```
Int* dynamicArray = new int[n];  // Size 'n' determined at runtime
delete[] dynamicArr;             // Must free memory manually
```

Arrays
- Dynamic Array vs. STL vector

Feature	Dynamic Array	STL vector
Memory Management	Manual (**new** and **delete**[])	Automatic
Resizing	Manual (requires reallocation)	Automatic and seamless
Ease of Use	Low (error-prone)	High (many utility features)
Performance	Slightly faster (no overhead)	Slightly slower (due to overhead)
Safety	No bounds checking	Safer with .at()

Arrays
- An example

```
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
};

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; }
```

Arrays
- An example

```
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;
  int numEntries;
  GameEntry* entries;
};
```

```
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;
}
```

Arrays
- 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
}
```

Arrays
- Object Removal

```
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
}
```

Arrays
- Sorting an Array (algorithm)

Algorithm InsertionSort(A):

Input: An array A of n comparable elements

Output: The array A with elements rearranged in nondecreasing order

for $i \leftarrow 1$ to $n-1$ **do**

  {Insert $A[i]$ at its proper location in $A[0], A[1], \ldots, A[i-1]$}

  $cur \leftarrow A[i]$

  $j \leftarrow i-1$

  while $j \geq 0$ and $A[j] > cur$ **do**

    $A[j+1] \leftarrow A[j]$

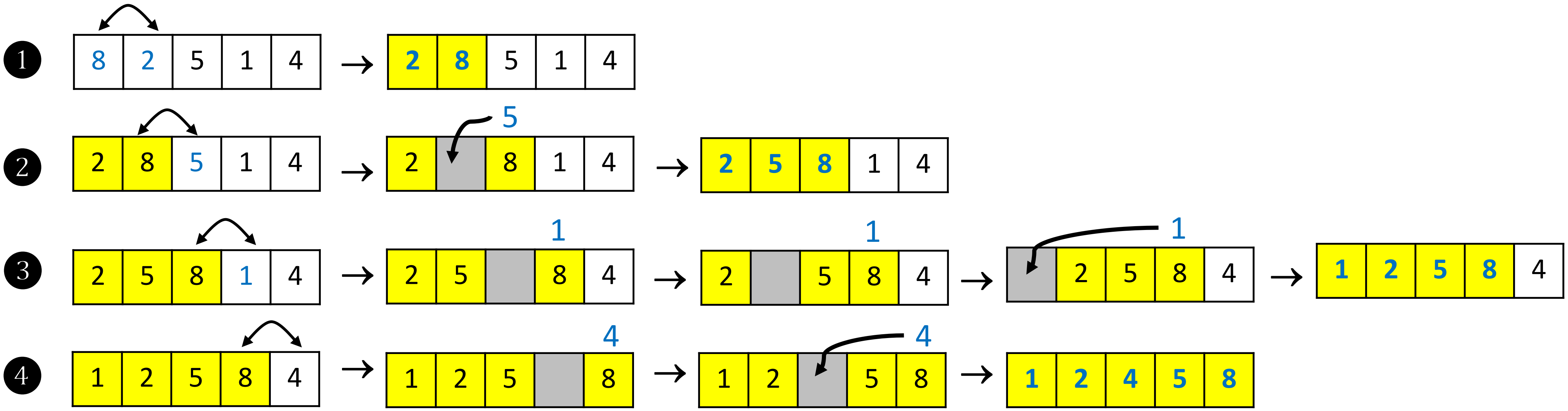
    $j \leftarrow j-1$

  $A[j+1] \leftarrow cur$ {cur is now in the right place}

Arrays
- Sorting an Array (C++ Implementation)

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

Arrays
- Insertion-Sort



Arrays
- Matrix

- Two-Dimensional Arrays
  - **Static Array**

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

```
const int N_DAYS = 7;
const int N_HOURS = 24;
int schedule[N_DAYS][N_HOURS];
```

  - **Dynamic Array**

```
int** M = new int*[n];           // allocate an array of row pointers
for (int i = 0; i < n; i++)
  M[i] = new int[m];             // allocate the i-th row

for (int i = 0; i < n; i++)
  delete[] M[i];                 // delete the i-th row
delete[] M;                      // delete the array of row pointers
```

  - **STL vector**

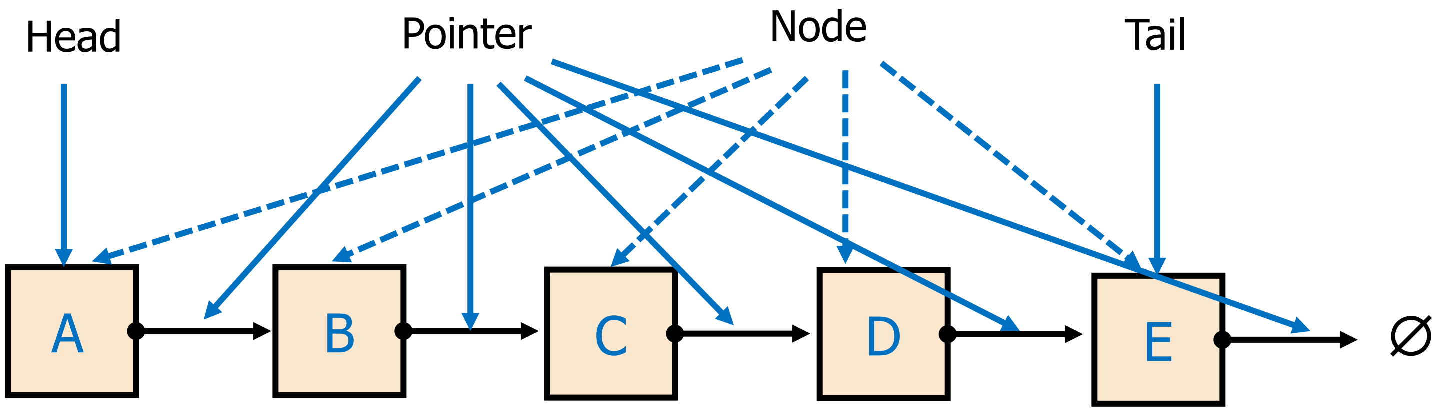
```
vector< vector<int> > M(n, vector<int>(m));
cout << M[i][j] << endl;
```

Linked List

- A linked list is a sequential list of nodes that hold data which point to other nodes also containing data
- Where are linked lists used?
  - Used in many List, Queue, and Stack implementations
  - Great for creating circular lists
  - Can easily model real world objects such as trains
  - Used in separate chaining, which is present certain Hashtable implementations to deal with hashing collisions
  - Often used in the implementation of adjacency lists for graphs

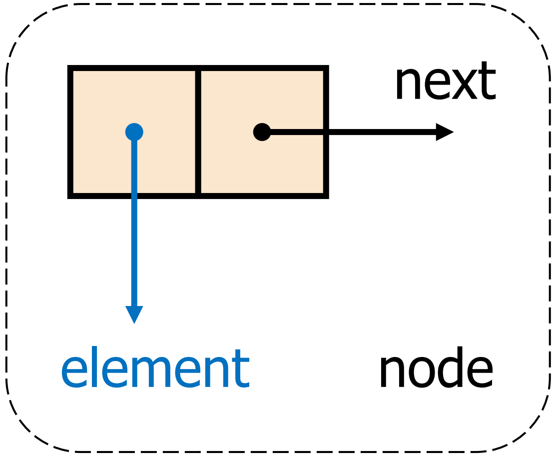
Linked List
- Terminology

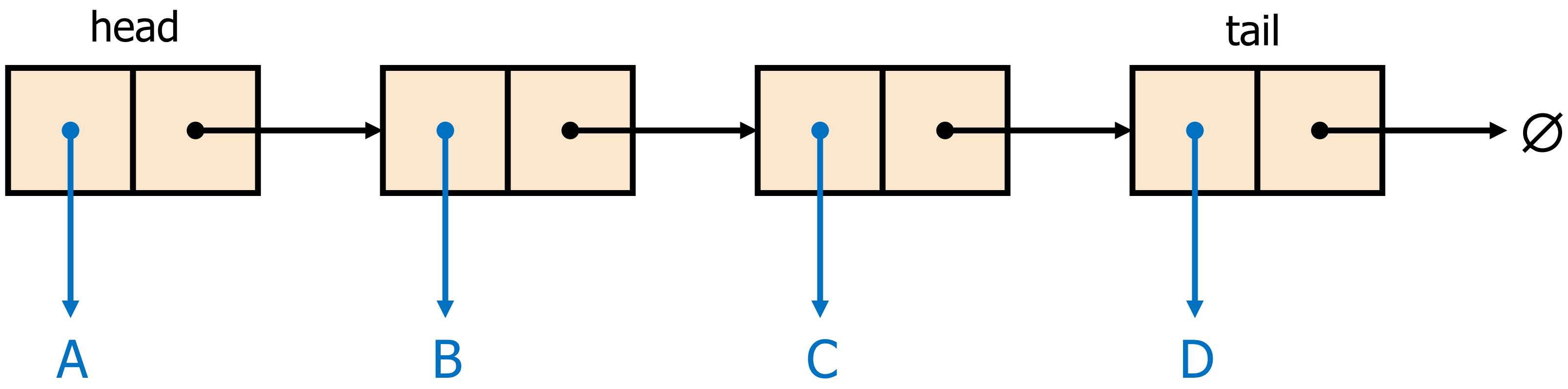
- Head : The first node in a linked list
- Tail : The last node in a linked list
- Pointer : Reference to another node
- Node : An object containing data and pointer(s)



Singly Linked Lists

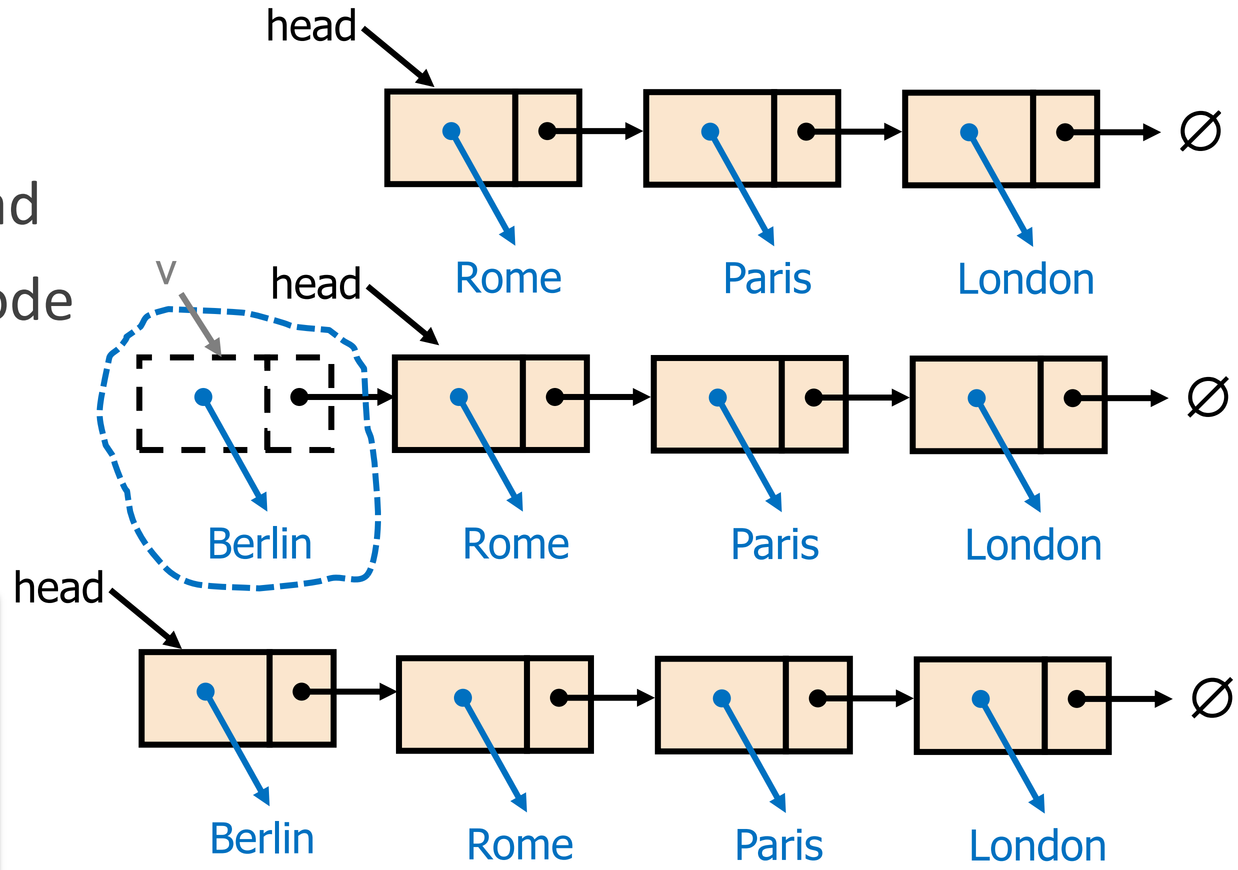
- A Singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - Element
  - Link to the next node





Singly Linked Lists
- Inserting at the Head

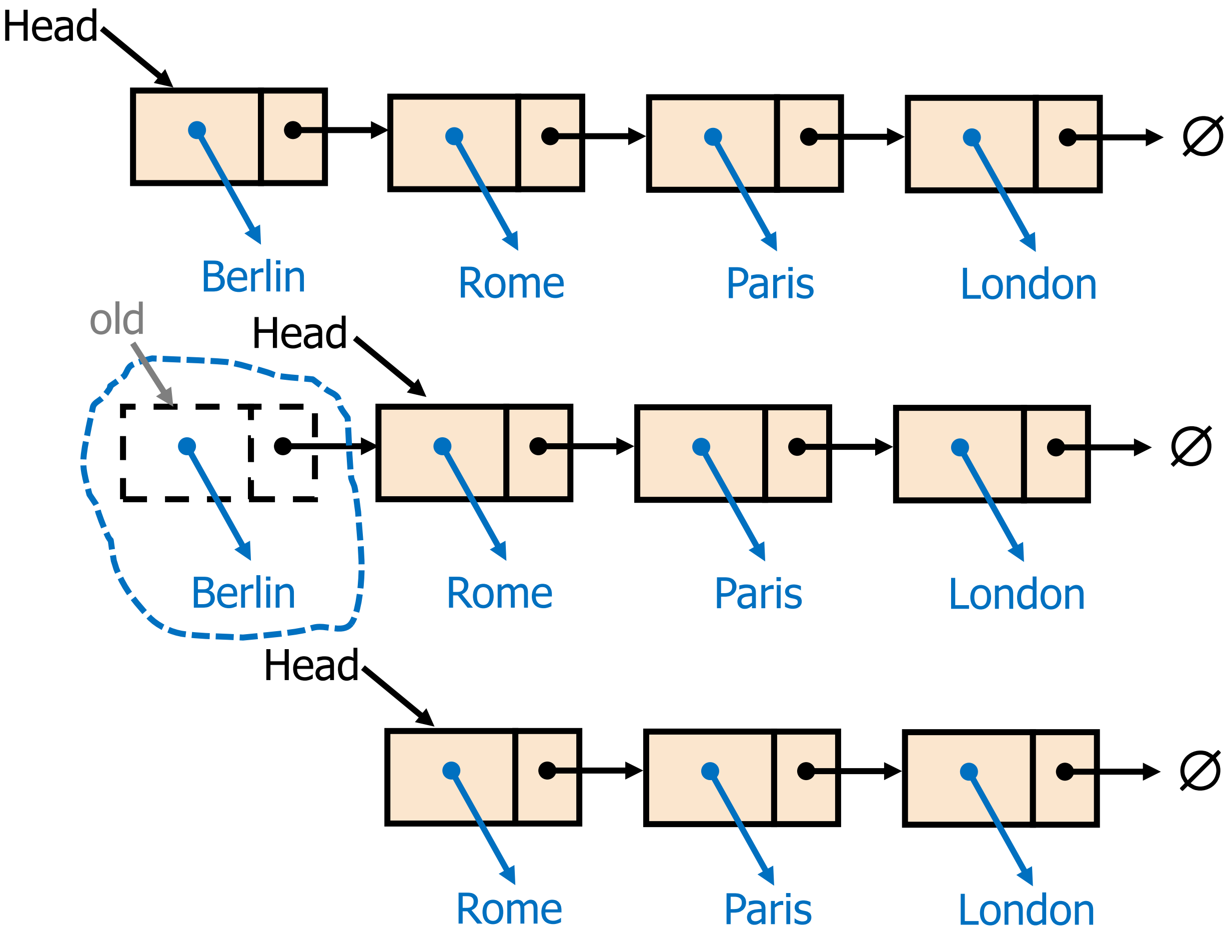
1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node




```
void StringLinkedList::addFront(const string& e) {
  StringNode* v = new StringNode;
  v->elem = e;
  v->next = head;
  head = v;
}
```

Singly Linked Lists
- Removing at the Head

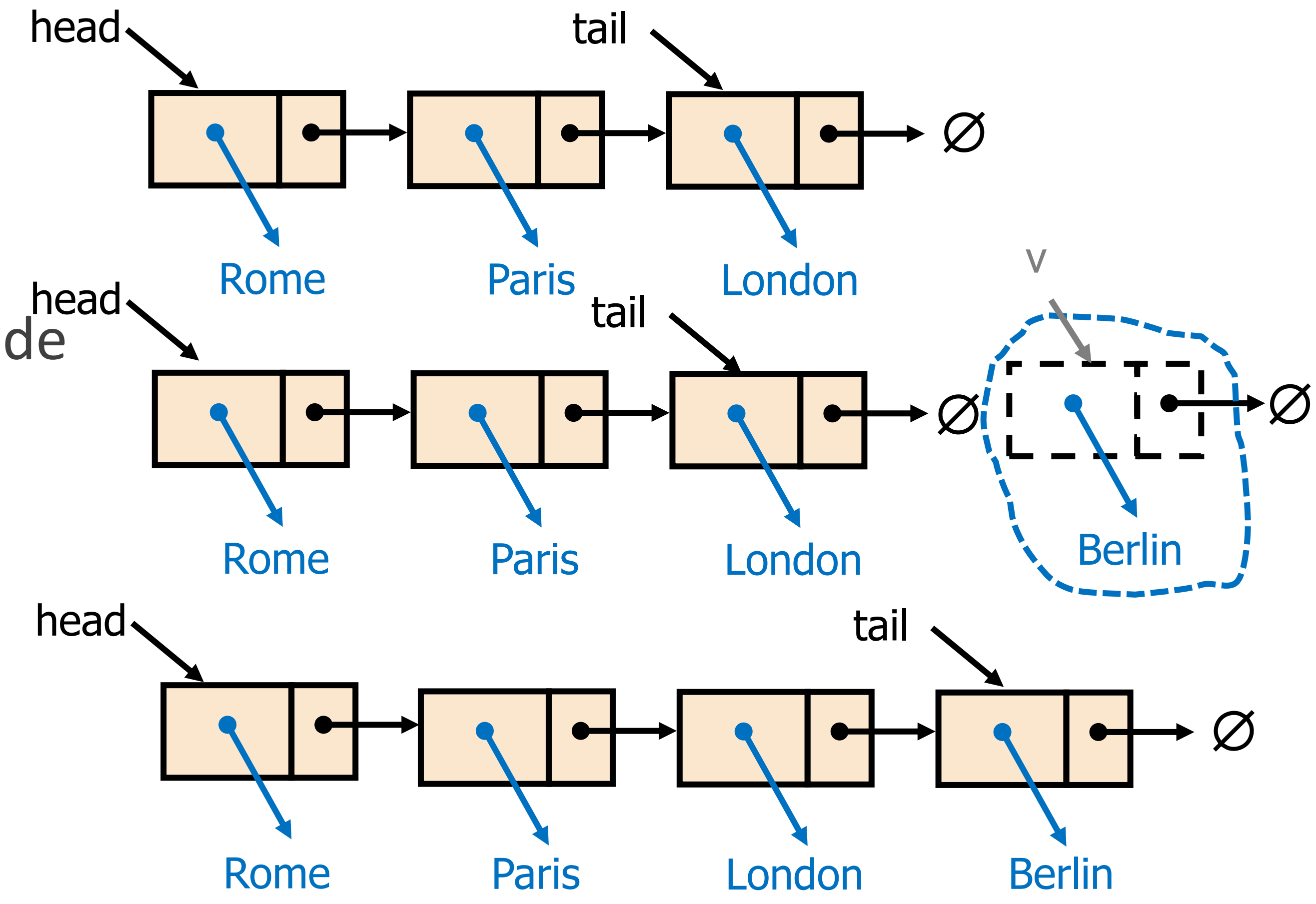
1. Update head to point to next node in the list
2. Allow garbage collect to reclaim the former first node




```
void StringLinkedList::removeFront() {
  StringNode* old = head;
  head = old->next;
  delete old;
}
```

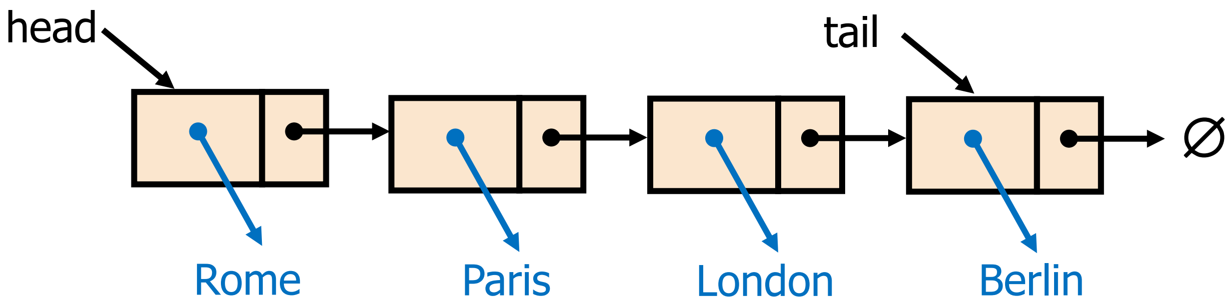
Singly Linked Lists
- Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



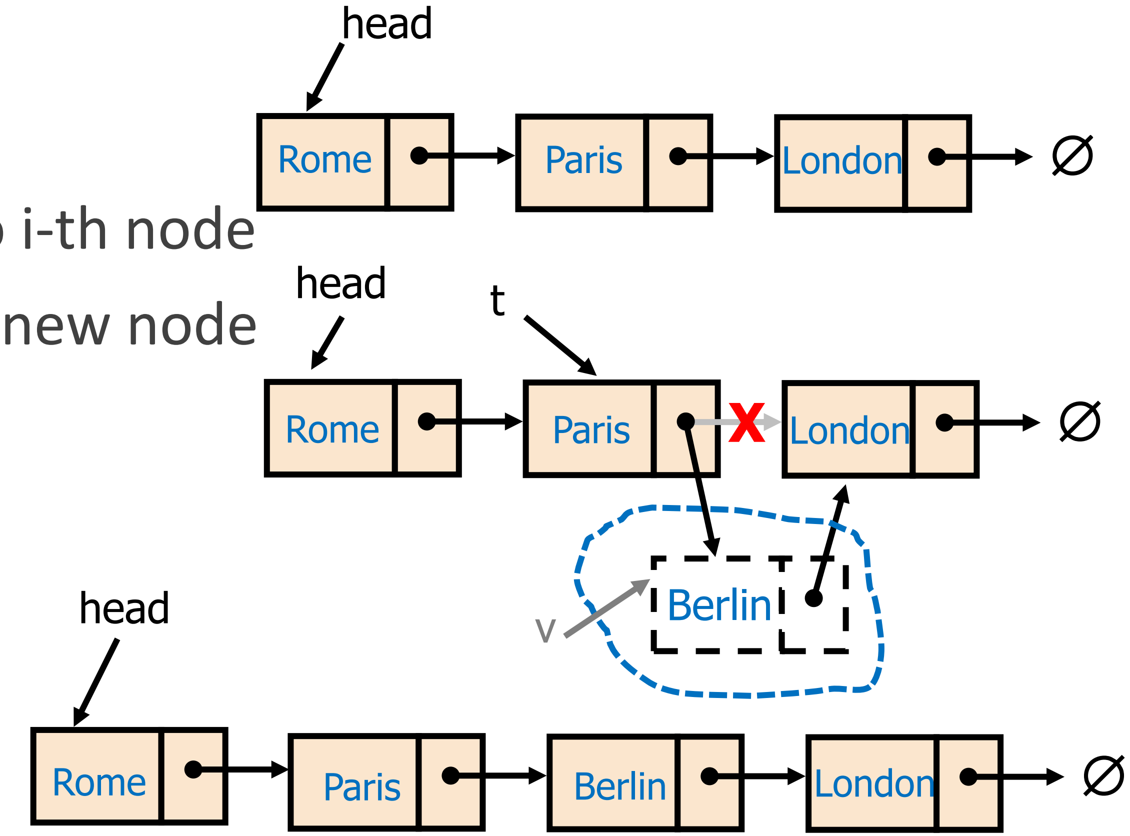
Singly Linked Lists
- Removing at the Tail

- Removing at the tail of a singly linked list is not efficient
- There is no constant-time way to update the tail to point to the previous node



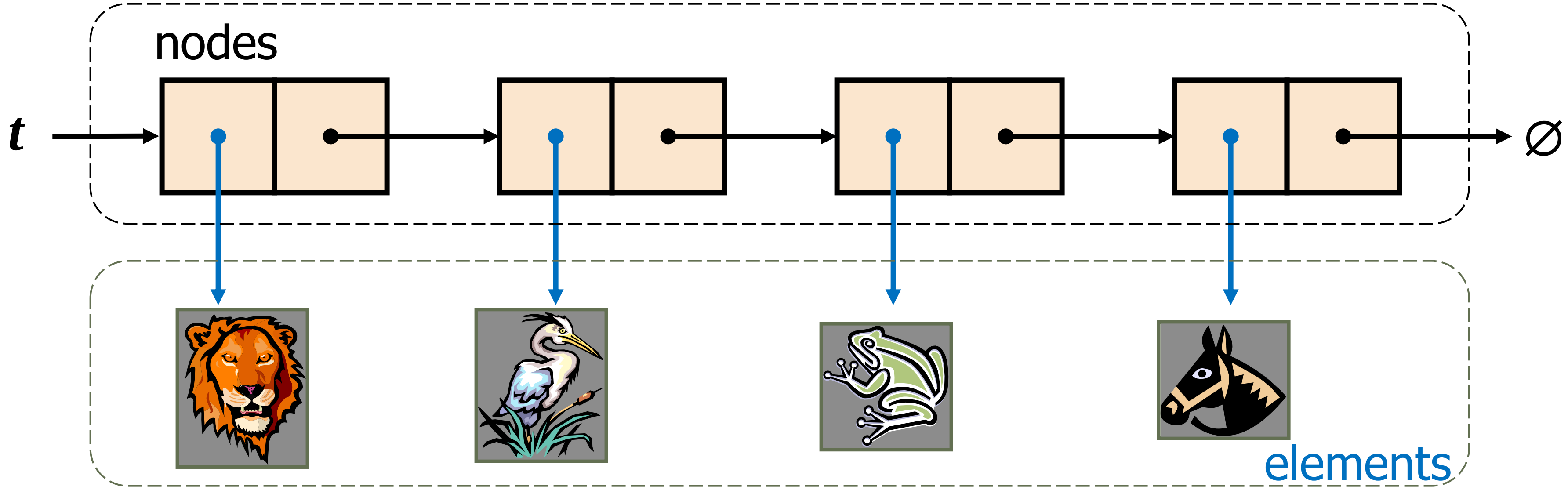
Singly Linked Lists
- Inserting at i-th node

1. Allocate a new node
2. Insert new element
3. Update new node's next link point to i-th node
4. Update i-th node's prev link point to new node



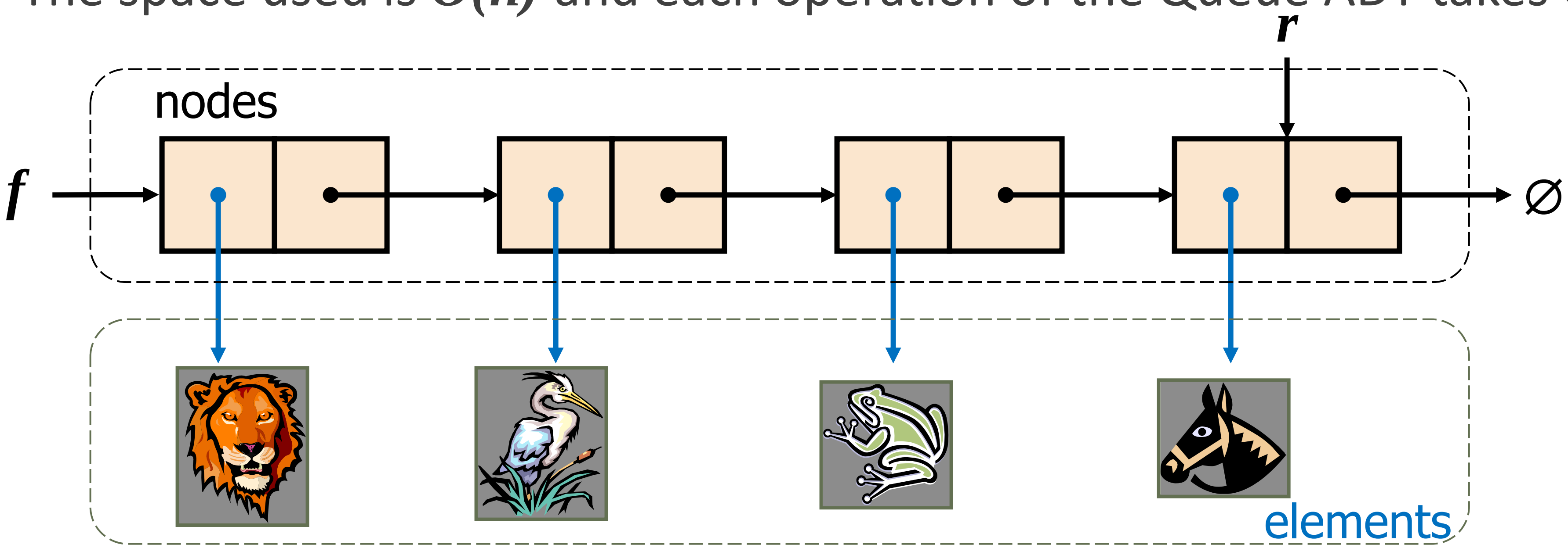
Stack as a Singly Linked Lists

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



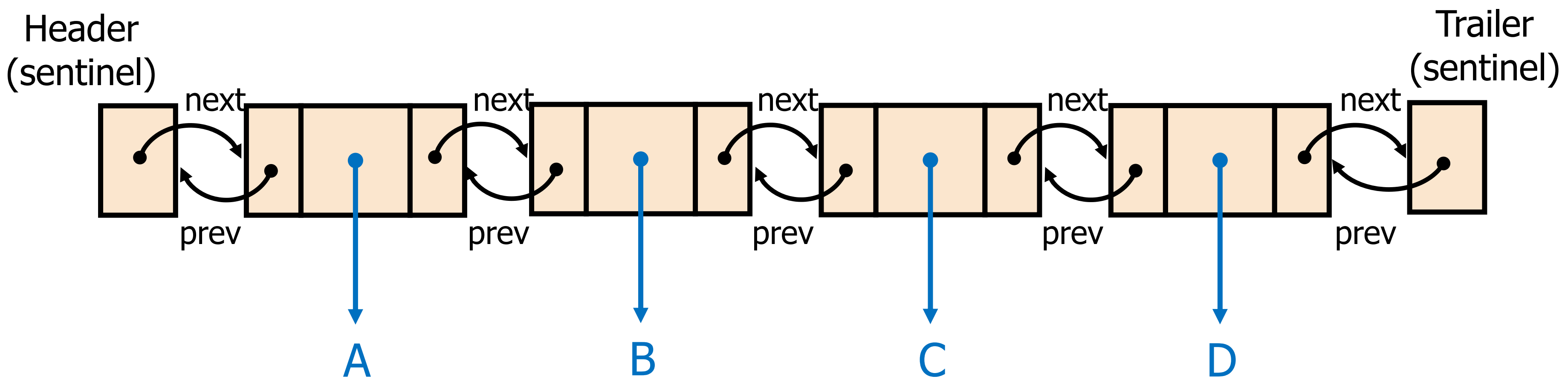
Queue as a Singly Linked Lists

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time

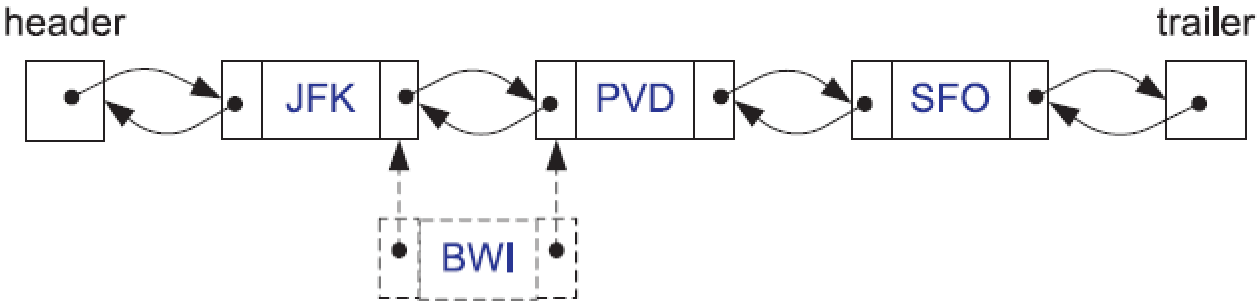


Doubly Linked Lists

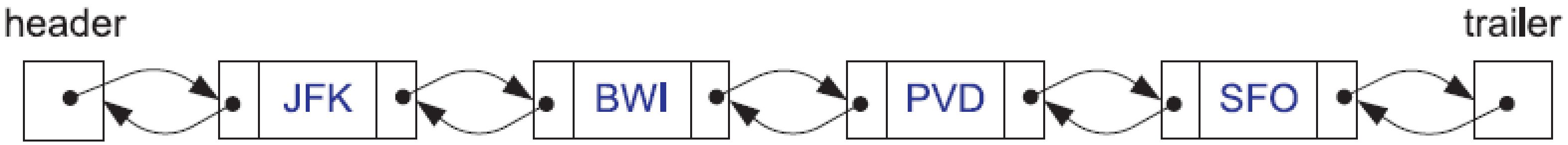
- Why use a Doubly Linked List?
  - Limitations of Singly Linked List
    : Time consuming to remove any node other than the head
  - Allow movement in both directions (forward and reverse)
    : Store two pointers (next and prev link)



Doubly Linked Lists
- Insertion

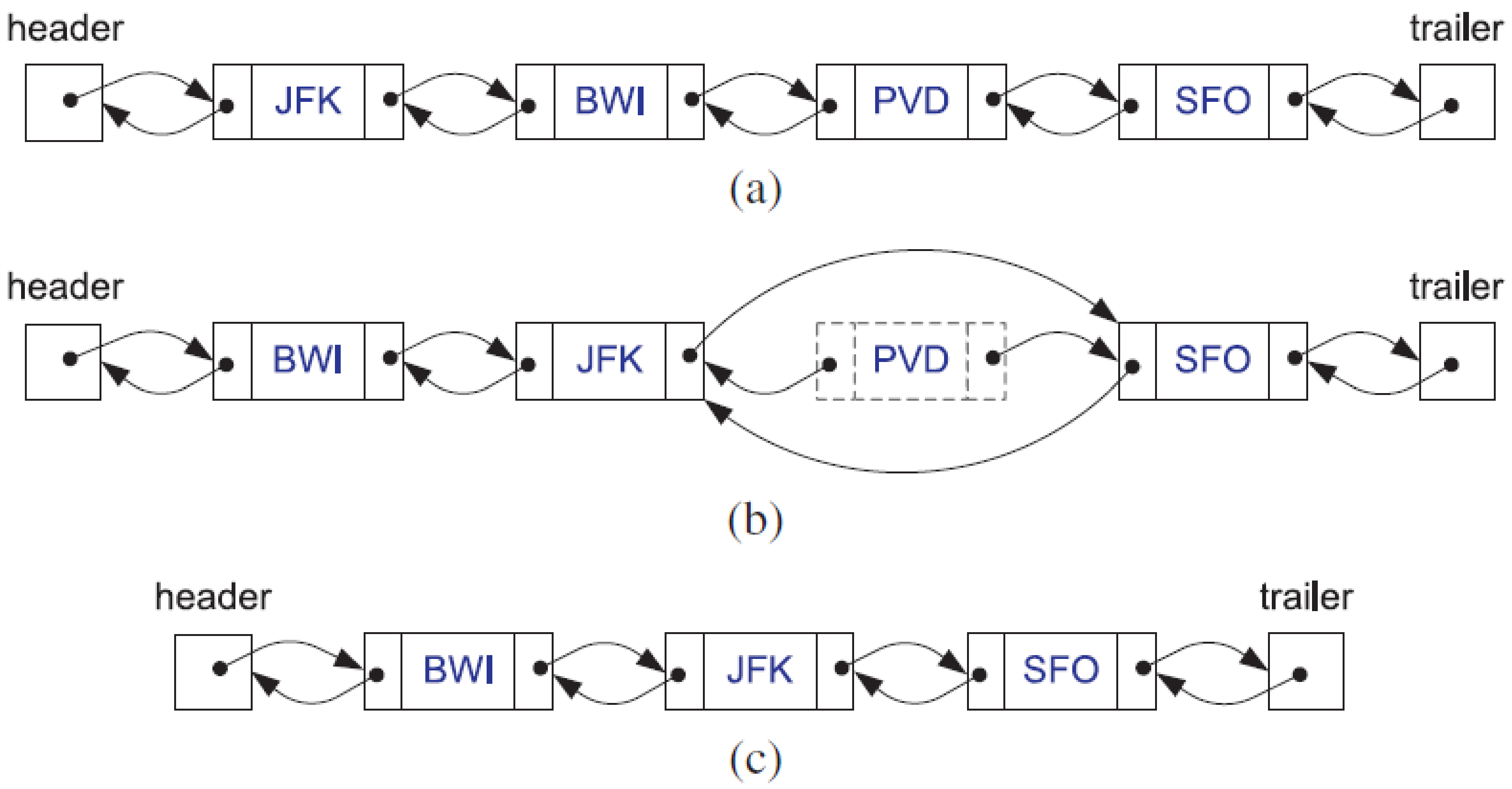


(a)



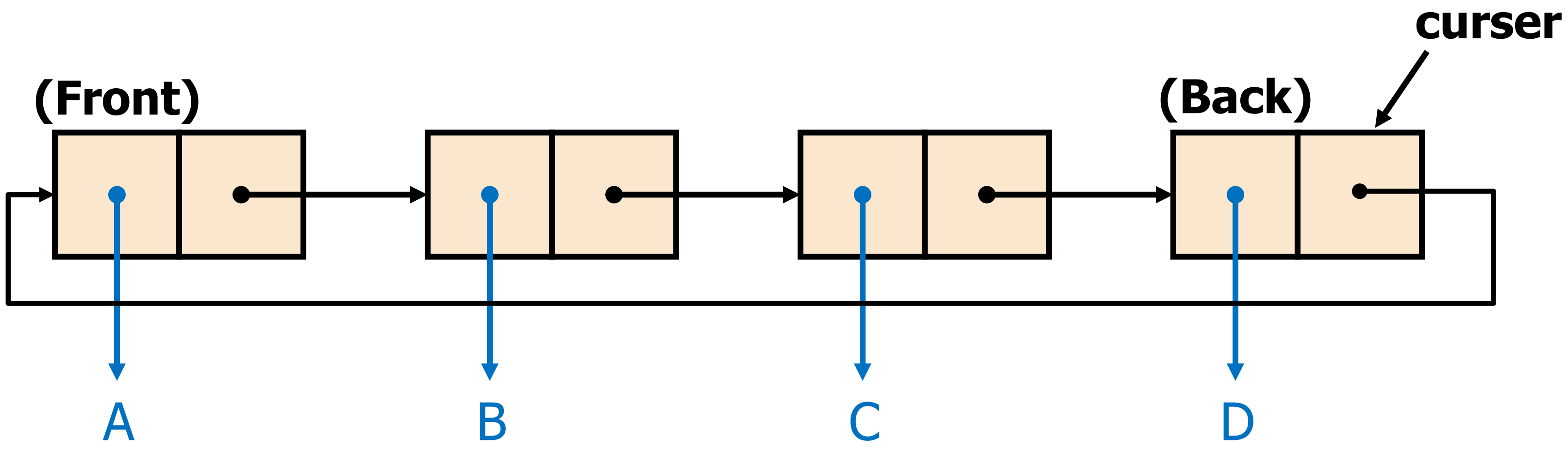
(b)

Doubly Linked Lists
- Removal



Circularly Linked Lists

- Characteristics of Circularly Linked Lists
  - No specific head or tail
  - A special node, called the **cursor**, is used as a starting point for traversal
    - **Front** : the element immediately following the cursor
    - **Back** : the element at the cursor node
  - If the circular link is cut, the list becomes a regular singly linked list



Circularly Linked Lists
- Reversing a Linked List

```
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);
  }
}
```

Recursion
- The Recursion Pattern

- Recursion : when a method calls itself
- Classic example—the factorial function:
  - n! = 1·2·3···(n-1)·n
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- As a C++ method:

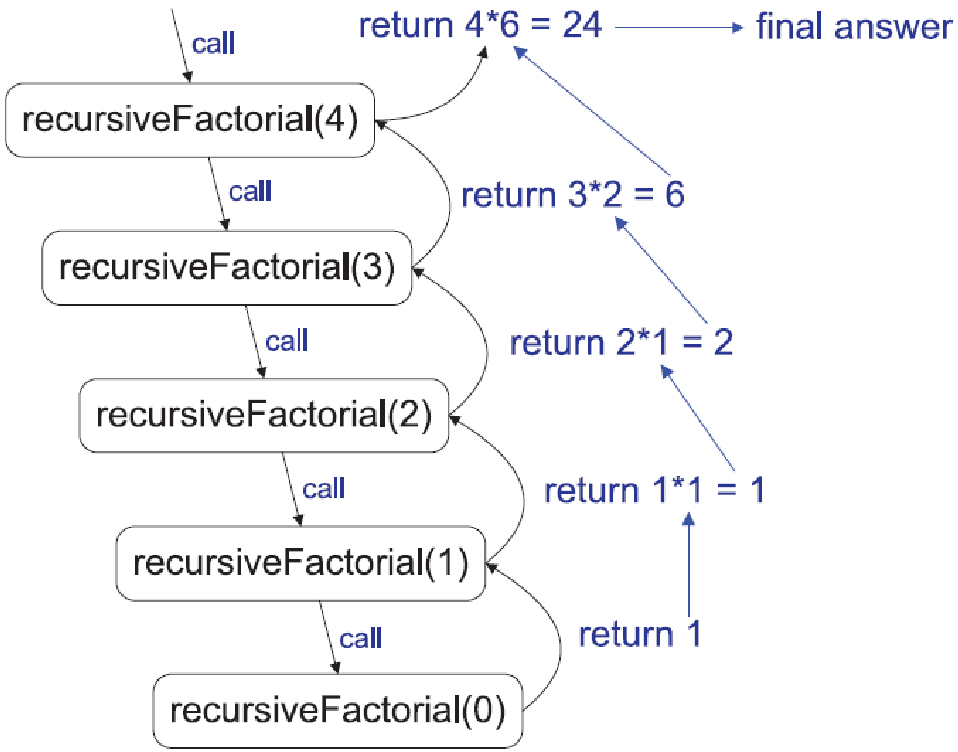
```
int recursiveFactorial(int n) {                    // recursive factorial function
  if (n == 0) return 1;                            // basis case
  else return n * recursiveFactorial(n−1);         // recursive case
}
```

Recursion
- Content of a Recursive Method

- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case)
  - Every possible chain of recursive calls must eventually reach a base case
- Recursive calls
  - Calls to the current method
  - Each recursive call should be defined so that it makes progress towards a base case

Recursion
- Visualizing Recursion

- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value
- Example



Recursion
- Linear Recursion

- Test for base cases
  - Every possible chain of recursive calls must eventually reach a base case
- Recur once
  - Perform a single recursive call
  - Might branch to one of several possible recursive calls
  - makes progress towards a base case

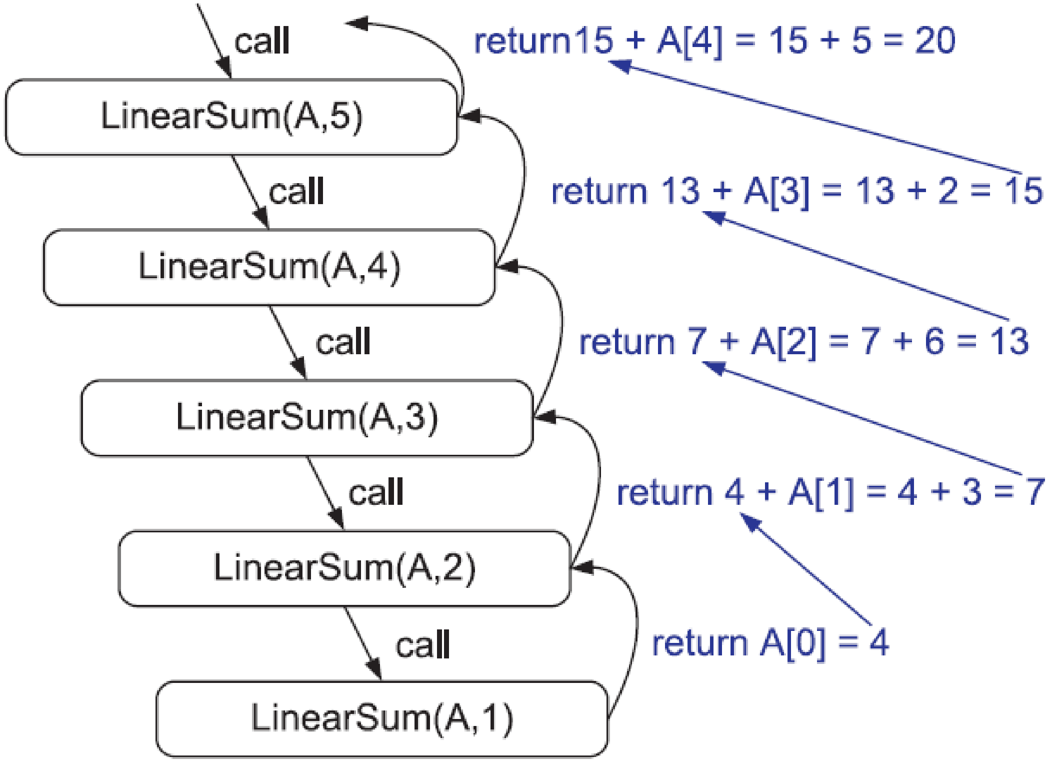
Recursion
- Example of Linear Recursion

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$(A, n-1) + A[n-1]$



Recursion
- Reversing an Array

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

Recursion
- Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursive
- This sometimes requires we define additional parameters that are passed to the method
- For example, we defined the array reversal method as ReverseArray(A,I,j), not ReverseArray(A)

Recursion
- Tail Recursion

- A linearly recursive method makes its recursive call as its last step
- The array reversal method is an example
- Such methods can be easily converted to non-recursive methods (which saves on some resources)

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

Recursion
- Binary Recursion

- Binary recursion occurs whenever there are two recursive calls for each non-base case
- Example: the DrawTicks method for drawing ticks on an English ruler

```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

(a)

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

(b)

```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
```

(c)

Recursion

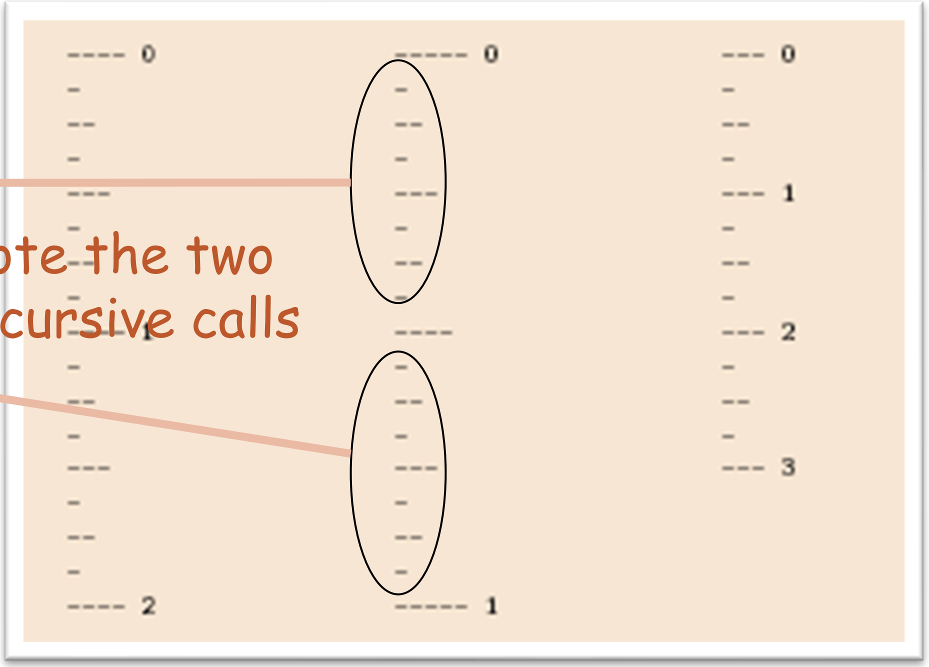
- A Binary Recursive Method for Drawing Ticks

```
// draw a tick with no label
public static void drawOneTick(int tickLength) { drawOneTick(tickLength, - 1); }
  // draw one tick
public static void drawOneTick(int tickLength, int tickLabel) {
   for (int i = 0; i < tickLength; i++)
      System.out.print("-");
   if (tickLabel >= 0) System.out.print(" " + tickLabel);
   System.out.print("\n");
}
public static void drawTicks(int tickLength) { // draw ticks of given length
   if (tickLength > 0) {               // stop when length drops to 0
      drawTicks(tickLength- 1);        // recursively draw left ticks
      drawOneTick(tickLength);         // draw center tick
      drawTicks(tickLength- 1);        // recursively draw right ticks
   }
}
public static void drawRuler(int nInches, int majorLength) { // draw ruler
   drawOneTick(majorLength, 0);        // draw tick 0 and its label
   for (int i = 1; i <= nInches; i++)  {
      drawTicks(majorLength- 1);       // draw ticks for this inch
      drawOneTick(majorLength, i);     // draw tick i and its label
   }
}
```

drawTicks(length)

Input: length of a 'tick'

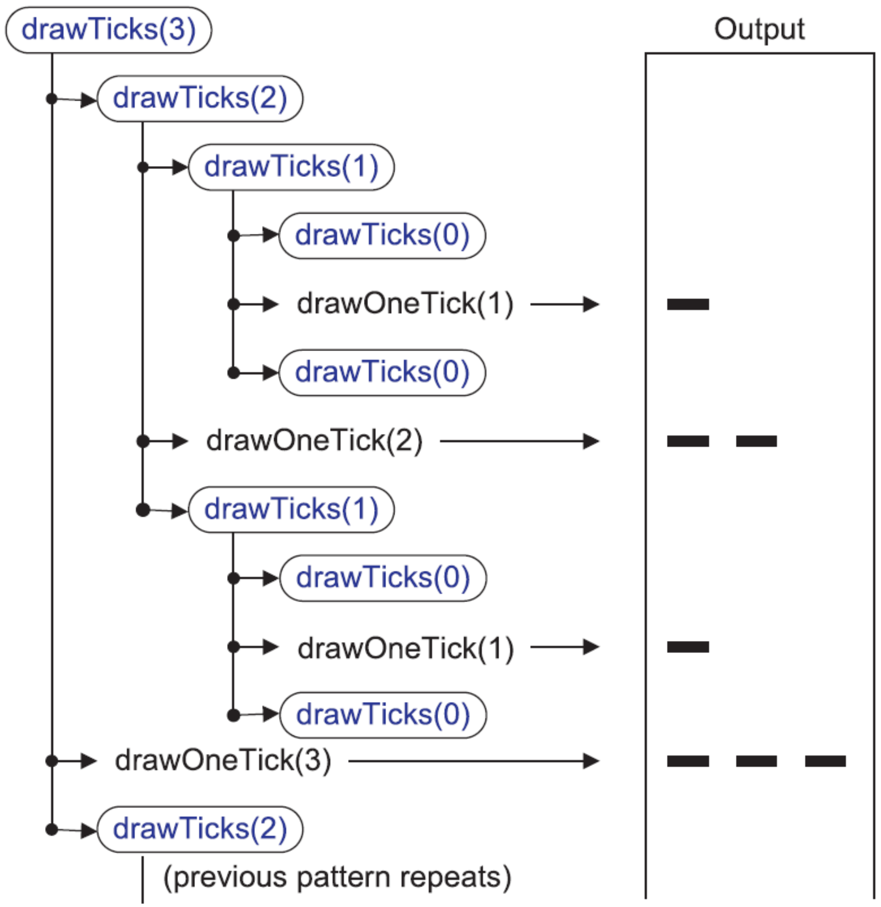
Output: ruler with tick of the given length in the middle and smaller rulers on either side



Note the two recursive calls

Recursion
- Recursive Drawing Method

- The drawing method is based on the following recursive definition
- An interval with a central tick length L $\geq$1 consists of:
  - An interval with a central tick length L–1
  - An single tick of length L
  - An interval with a central tick length L–1

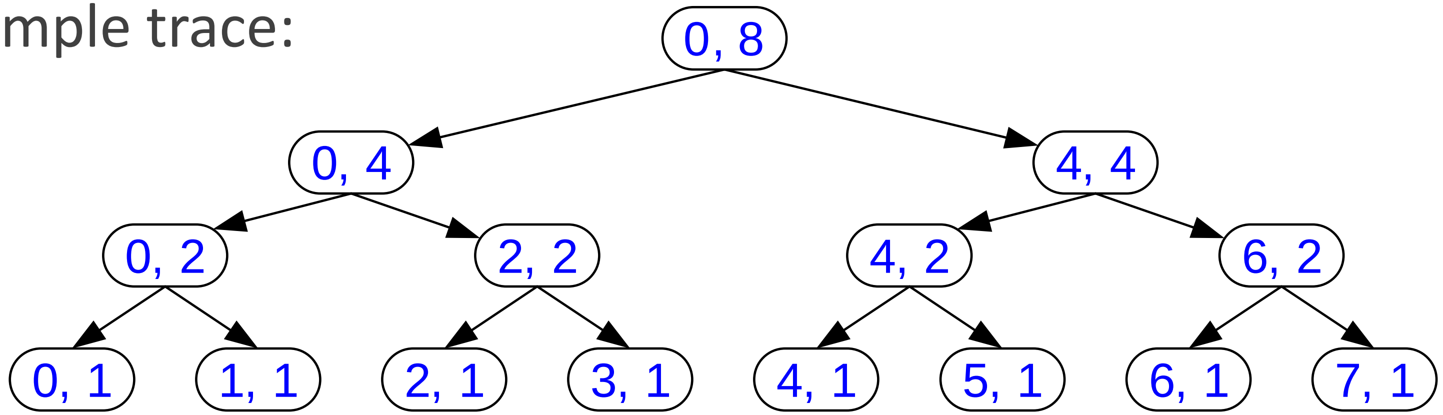


Recursion
- Another Binary Recursive Method

- Problem: add all the numbers in an integer array A:

  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($A, i, n/2$) + BinarySum($A, i + n/2, n/2$)

- Example trace:



Recursion
- Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1$

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):

Input: Nonnegative integer k

Output: The kth Fibonacci number F_k

if $k = 1$ **then**

return k

else

return BinaryFib(k - 1) + BinaryFib(k - 2)

Recursion
- Fibonacci Algorithm Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

Recursion
- A Better Fibonacci Algorithm

- Use linear recursion instead

  Algorithm LinearFibonacci(k):

  Input: A nonnegative integer k

  Output: Pair of Fibonacci numbers (F_k , F_{k-1})

  if $k = 1$ **then**

  return (k, 0)

  else

  (i, j) = LinearFibonacci($k - 1$)

  return (i +j, i)

- LinearFibonacci makes k–1 recursive calls

Recursion
- Multiple Recursion

- Motivating example:
  - summation puzzles
    - pot + pan = bib
    - dog + cat = pig
    - boy + girl = baby

p=4, o=2, t=1. a=3, n=7, b=8, i=5	421 + 437 = 858
d=1, o=2, g=8. c=6, a=3, t=0, p=7, i=5	128 + 630 = 758
b=7, o=2, y=8. g=6, i=4, r=5, l=0, a=1	728 + 6450 = 7178

- Multiple recursion:
  - makes potentially many recursive calls
  - not just one or two

Recursion
- Algorithm for Multiple Recursion

Algorithm PuzzleSolve(k, S, U):

Input: An integer k, sequence S, and set U

Output: An enumeration of all k-length extensions to S using elements in U without repetitions

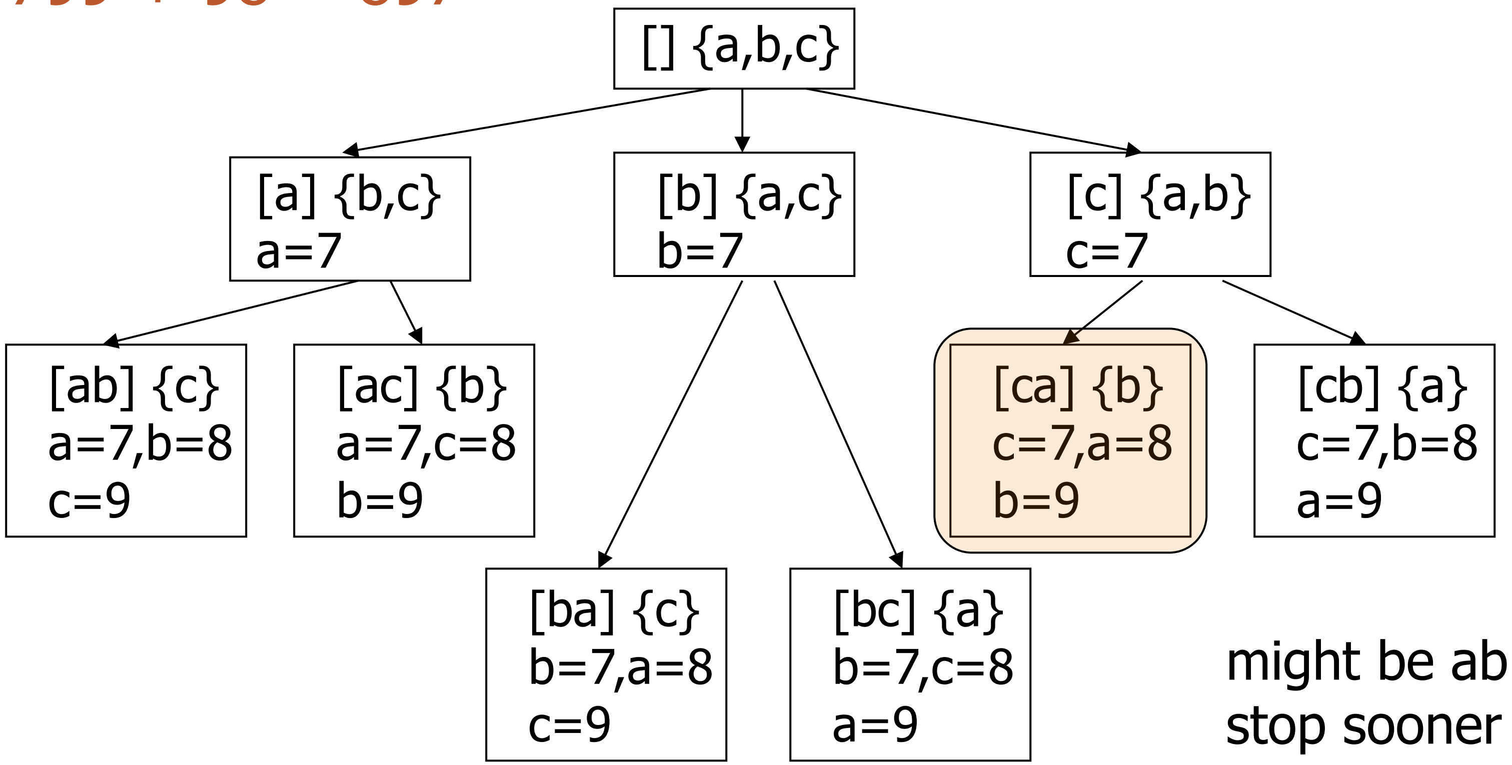
```
for each e in U do
    Remove e from U {e is now being used}
    Add e to the end of S
    if k = 1 then
        Test whether S is a configuration that solves the puzzle
        if S solves the puzzle then
            return "Solution found: " S
    else
        PuzzleSolve(k − 1, S, U)
    Add e back to U {e is now unused}
    Remove e from the end of S
```

Recursion
- Example for Multiple Recursion

cbb + ba = abc

799 + 98 = 897

a,b,c stand for 7,8,9; not necessarily in that order



might be able to stop sooner

Recursion
- Visualizing PuzzleSolve