IPB University
Bogor Indonesia
Department of
Computer Science
http://cs.ipb.ac.id/

Kuliah 1 Struktur Data

Konsep Struktur Data

Disampaikan oleh : Dr. Eng. Annisa, SKom, MKom

Outline

Pertemuan	Tinjauan Instruksional Khusus	Topik	Sub Topik	Pengajar
1	Mahasiswa dapat menjelaskan teknik abstraksi data dan representasi struktur data	Pendahuluan, Penjelasan kontrak kuliah dan overview	Algoritma dan Struktur data, Konsep struktur data, tipe data dasar, tipe data bentukan, Big-O	ANN (K1, K2) HRW (K3)

Data

da·ta

/ˈdadə,ˈdādə/

See definitions in:

All | Computing | Philosophy

noun

noun: **data**

facts and statistics collected together for reference or analysis.
"there is very little data available"

Similar: facts | figures | statistics | details | particulars | specifics | features

- the quantities, characters, or symbols on which operations are performed by a computer, being stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.
- PHILOSOPHY
  things known or assumed as facts, making the basis of reasoning or calculation.

Example : $C = A + B$

Data VS Information

- Example :
  - Data : ananab ekil I ← Collection of character

Arrange it in some ways (Ex: reverse)

  - Information : I like banana

If data is arranged in a systematic way then it gets a structure and become meaningful.

Struktur

struc·ture

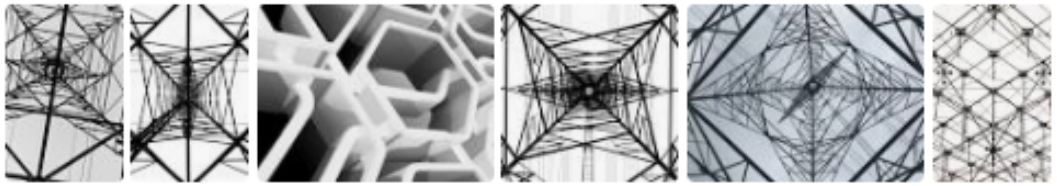
/ˈstrək(t)SHər/

noun

noun: **structure**; plural noun: **structures**

the arrangement of and relations between the parts or elements of something complex.
"flint is extremely hard, like diamond, which has a similar structure"

Similar: construction | form | formation | shape | composition | fabric



- a building or other object constructed from several parts.

  Similar: building | edifice | construction | erection | pile | complex

- the quality of being organized.
  "we shall use three headings to give some structure to the discussion"

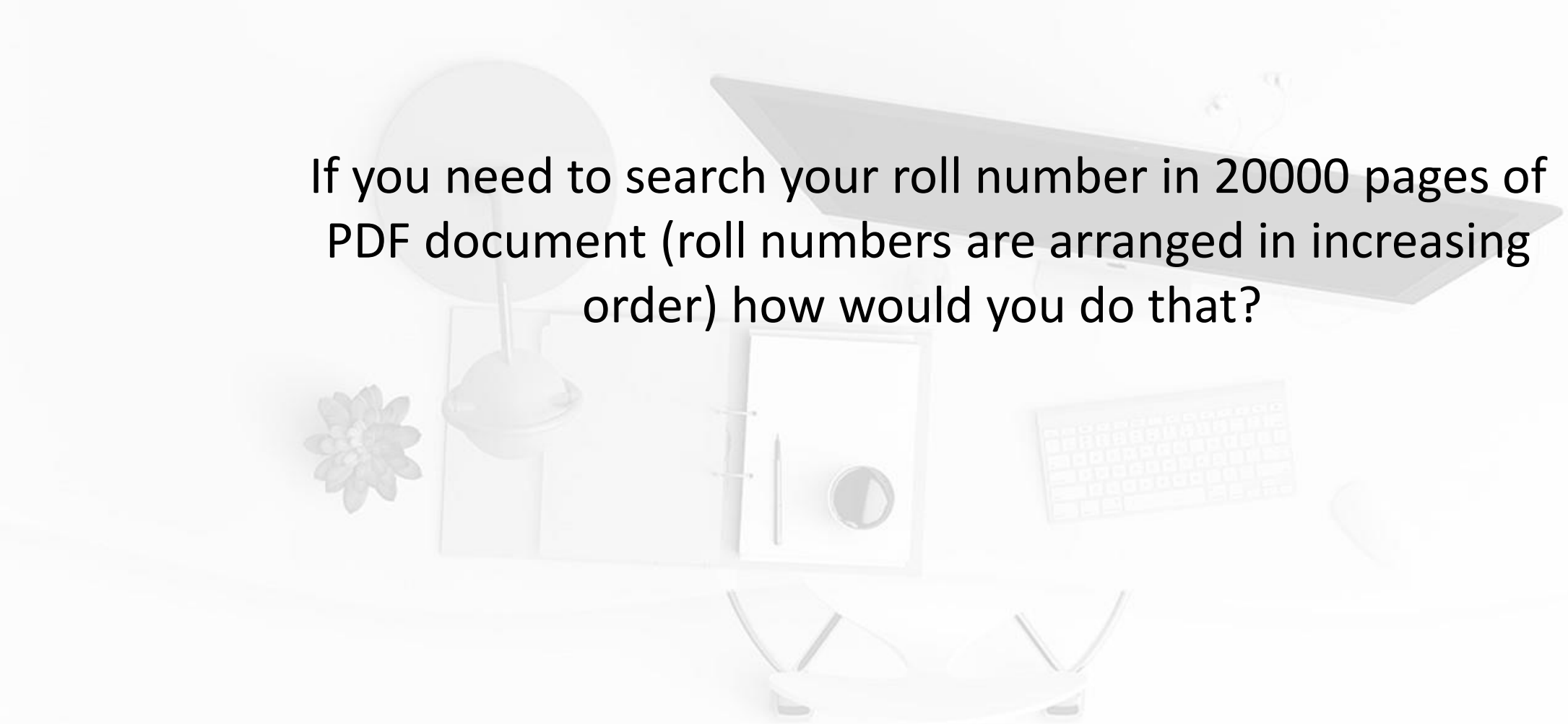
verb

verb: **structure**; 3rd person present: **structures**; past tense: **structured**; past participle: **structured**; gerund or present participle: **structuring**

construct or arrange according to a plan; give a pattern or organization to.
"the game is structured so that there are five ways to win"

Similar: arrange | organize | order | design | shape | give structure to

Why we need data structure?



If you need to search your roll number in 20000 pages of PDF document (roll numbers are arranged in increasing order) how would you do that?

Why We Need Data Structure ?

01

Case : Rolling Dice N times

Cetak berapa kali angka 1 keluar, angka 2 keluar, angka 3 keluar, berikut juga angka 4, 5, 6

02

Case : Pick N times of 500 numbers

Cetak berapa kali angka 1 keluar, angka 2 keluar, angka 3 keluar, berikut juga angka …..500

03

Case : Whose turn now?

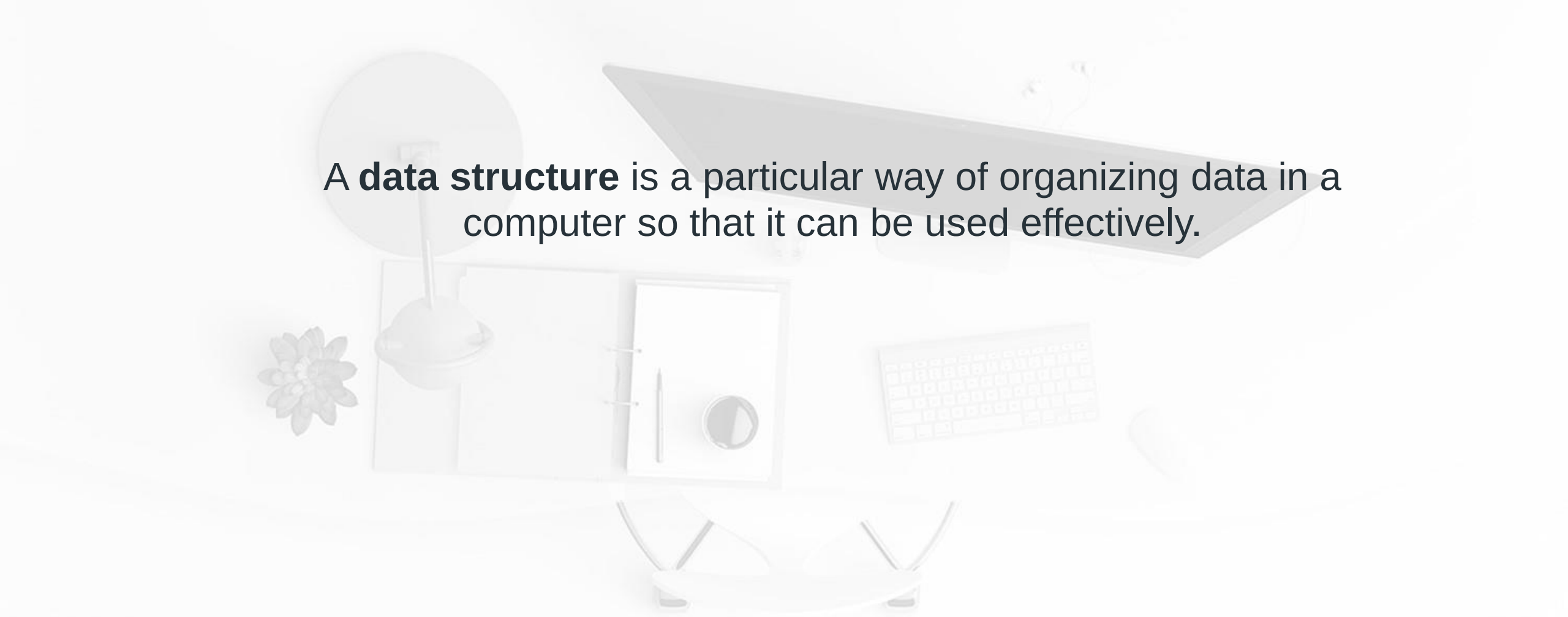
Sebuah printer harus mencetak N dokumen. Bagaimana mengatur agar setiap dokumen dapat tercetak tanpa ada yang merasa dilangkahi ?

04

Case : Searching sorted data

Carilah angka X dari sekumpulan angka yang telah terurut!

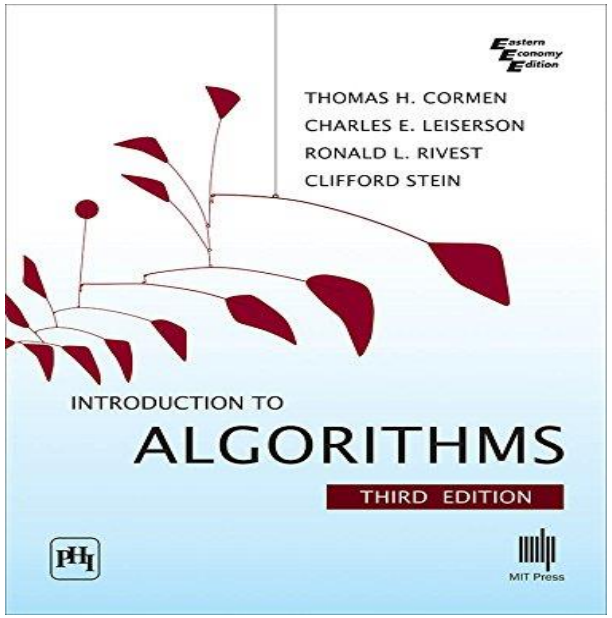
Data Structure



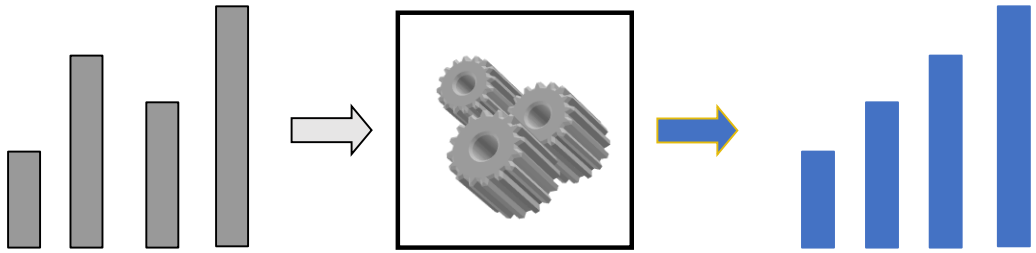
A **data structure** is a particular way of organizing data in a computer so that it can be used effectively.

Struktur Data dan Program

- Program mengolah data
  - Simak: data dosen, mahasiswa, laporan …
  - Game: skor pemain, status, posisi pemain ...
  - Website: hyperlink, isi web, log …
- Data perlu disimpan
  - Supaya bisa diolah, diproses dan didapatkan kembali
- Data perlu terstruktur
  - lebih mudah/efisien dalam mengakses dan mengolahnya
- (Struktur data + Algoritme) yg baik = Program yg Efisien



Algorithm!



Input

Algorithm 1
Algorithm 2
Algorithm 3
:
:
Algorithm n

Output

What is | Algorithm ?

An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

How to choose best algorithm?

Correctness | Efficiency

Bounded Resources!

Computers may be fast, but they are not infinitely fast

Memory may be cheap, but it is not free

Computing time and Memory Space are Bounded Resources :
Use them wisely

High Specs always faster?

B runs 20 times faster than A!

$$\frac{2 \cdot (10^6)^2 \text{ instructions}}{10^9 \text{ instructions/second}} = 2000 \text{ seconds}$$

Computer Name	A	B
One million numbers to sort	2000 seconds	100 seconds
Ten million numbers to sort	**2.3 days**	**Under 20 minutes**

$$\frac{50 \cdot 10^6 \lg 10^6 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 100 \text{ seconds}$$

Computer Name	A
Execution rate	One billion instructions per second
Algorithm	Insertion sort ($2n^2$)
Numbers to sort	one million

Computer Name	B
Execution rate	Ten million instructions per second
Algorithm	Merge sort ($50\, n \log n$)
Numbers to sort	one million

High specs computer doesn't help for slow algorithm and big amount of input !

Difficulties :

How are the algorithms coded?

Independent of a particular programming style

What computer should we use?

Independent of a particular computer specs

What data should the program use?

Independent of specific data

- *Each operation in an algorithm (or a program) has a cost.*
- *Each operation takes a certain of time*

The cost grows with the input size.

Analysis of Algorithms :

analyze the efficiency of different algorithms.

Analysis of Algorithms must be independent of specific implementations, computers, or data!

FIRST

Prove the correctness

Second

count the number of significant operations

Third

express the efficiency of algorithms using growth of functions (growth rates)

Efisiensi Program

- Faktor penting dalam merancang program:
  - Running time
- Bagaimana mengukur running time?

Jam	tergantung*
Jumlah input	tidak tergantung*

* Kecepatan mesin, sistem operasi, kualitas kompiler, bahasa pemrograman

Berapa Kompleksitasnya?

O(N)

```
read N
count := 0
for i := 1 to N do
        count := count + 1
end
```

O(N^2)

```
read N
count := 0
for i := 1 to N do
        for j := 1 to N do
                count := count + 1
        end
end
```

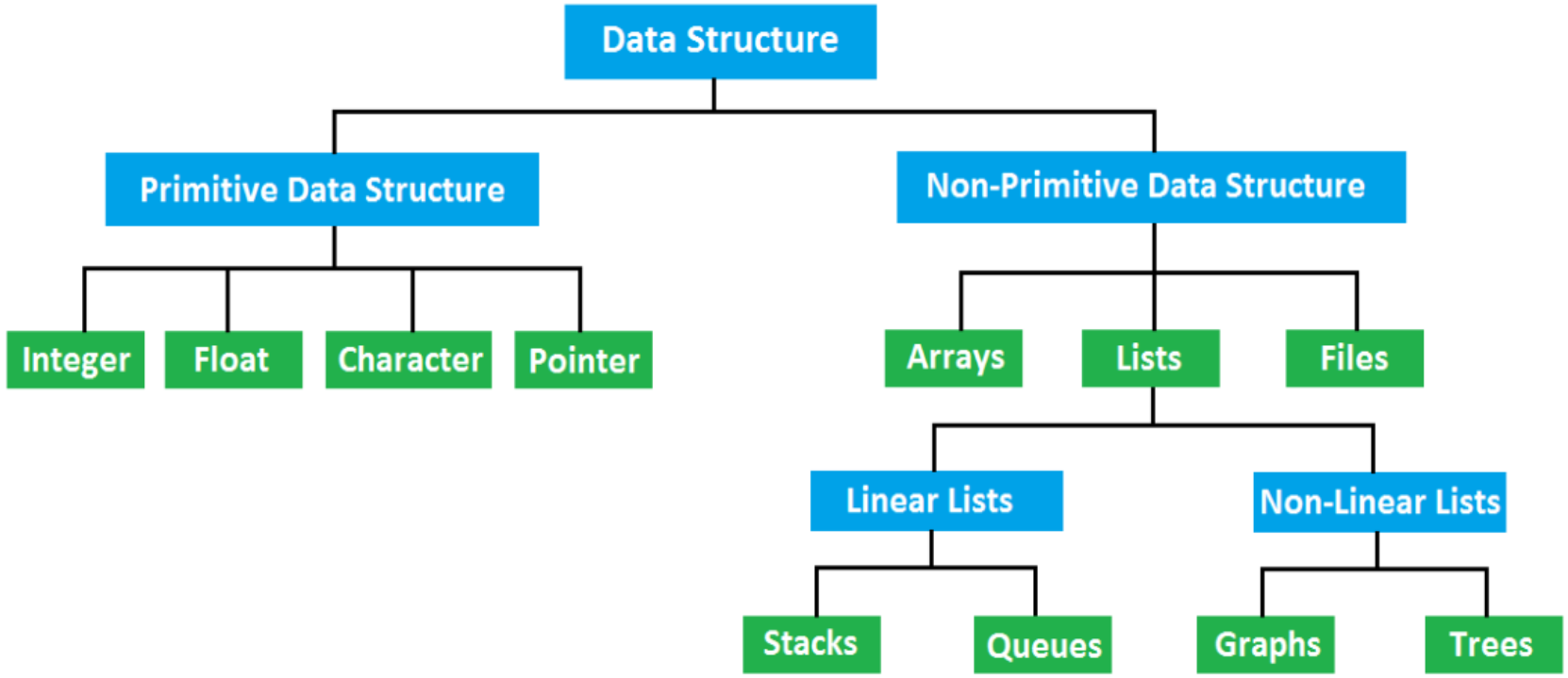
O(X+N)

```
read X
count := 0
for i := 1 to X do
        count := count + 1
end

read N
count := 0
for i := 1 to N do
        count := count + 1
end
```

O(N.(N+X))

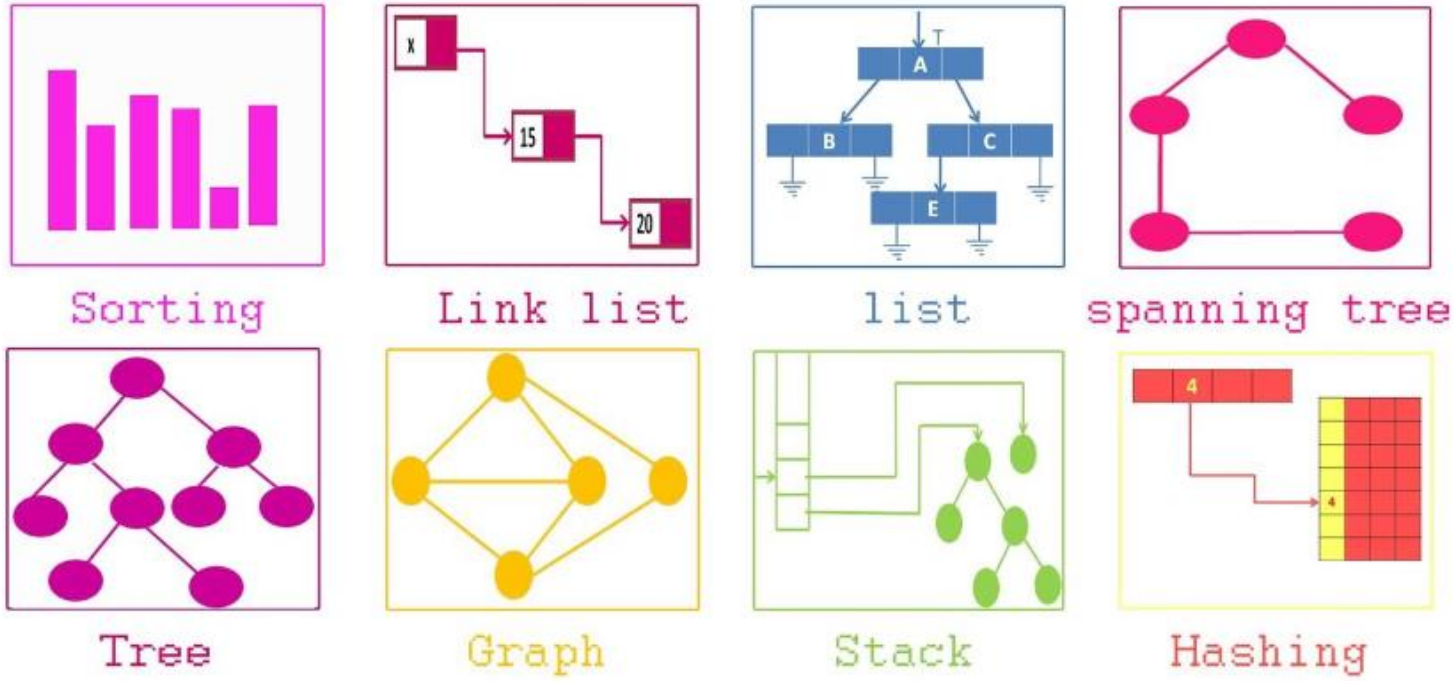
```
read N
read X
count := 0
for i := 1 to N do
        for j := 1 to N do
                count := count + 1
        end
        for j := 1 to X do
                count := count + 1
        end
end
```

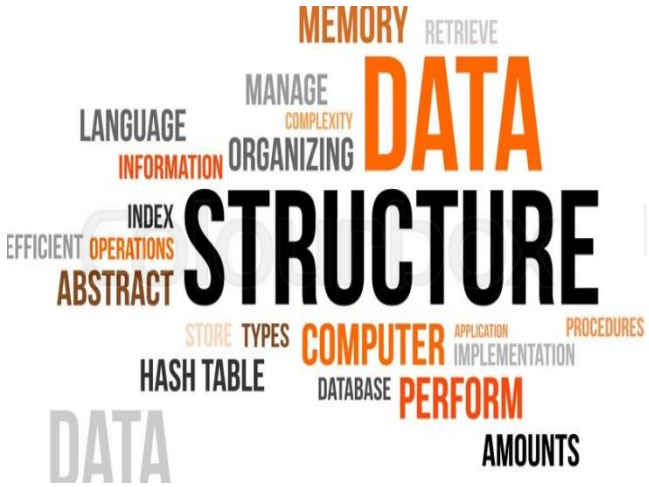
Data Structure Classification



- Define a certain domain of values
- Define set of operations allowed

Various Data Structure





All of Them use Data Structure

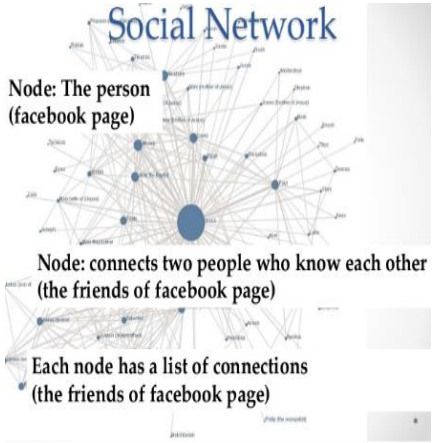
Semua aplikasi-aplikasi yang Anda ketahui menggunakan berbagai strukur data untuk mengolah seluruh data yang ada pada mereka secara efektif dan efisien.

Spelling Checker

Hashing, Trie, Ternary Search Tree

Chess Game

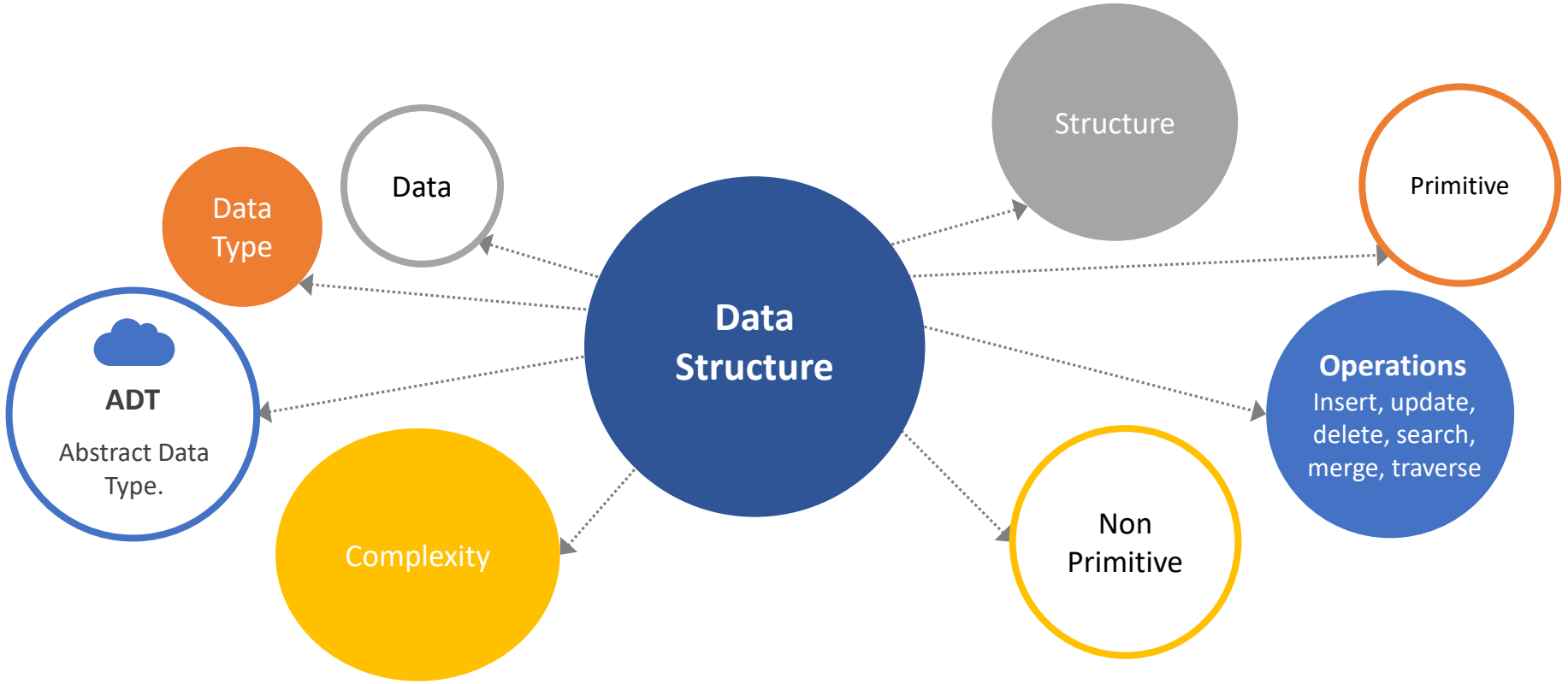
Hashing, Array, Dictionary



Social Network

Graph, tree, array, list

Terminologi



Tipe Data

- Data types atau tipe data adalah sebuah pengklasifikasian data berdasarkan jenis data, ukuran data, bagaimana ia disimpan, dan bagaimana dioperasikan
- Tipe data dibutuhkan agar kompiler dapat mengetahui bagaimana sebuah data akan digunakan.

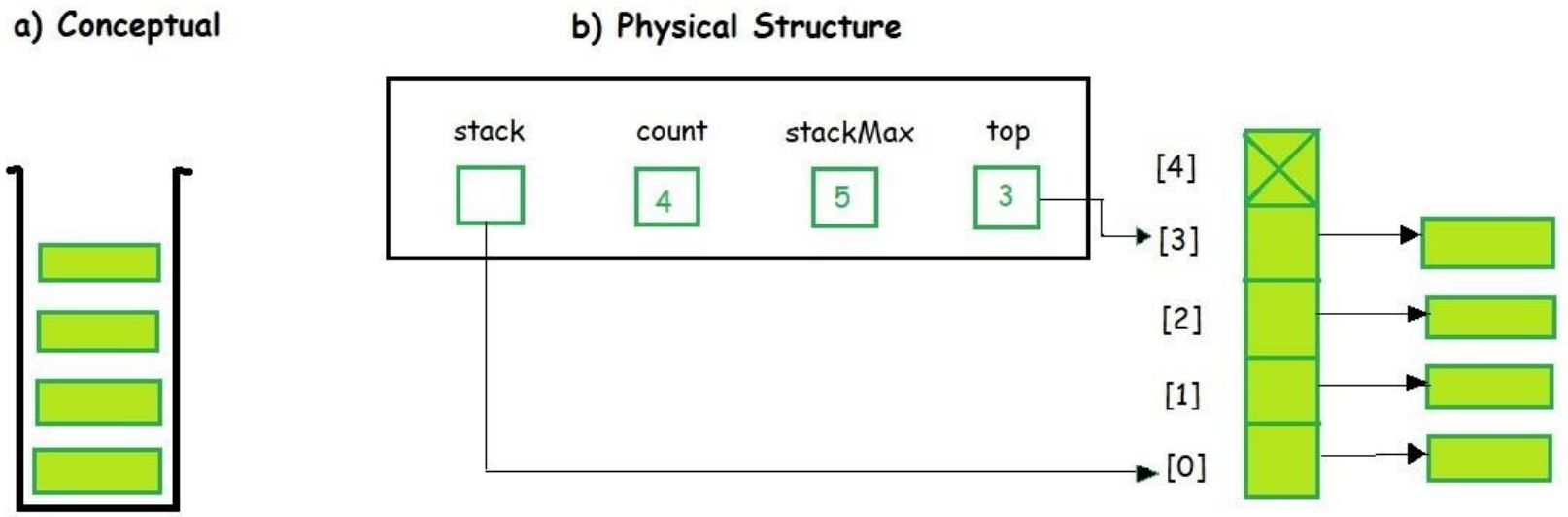
ADT

- Tipe data bentukan
- ADT adalah TYPE dan sekumpulan operasi dasar dari TYPE tersebut
- Definisi ADT hanya menyebutkan operasi apa yang akan dilakukan tetapi tidak bagaimana operasi ini akan dilaksanakan.
- Dalam C/C++ menggunakan typedef

ADT : Stack

```
//Stack ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *link;
} StackNode;
```

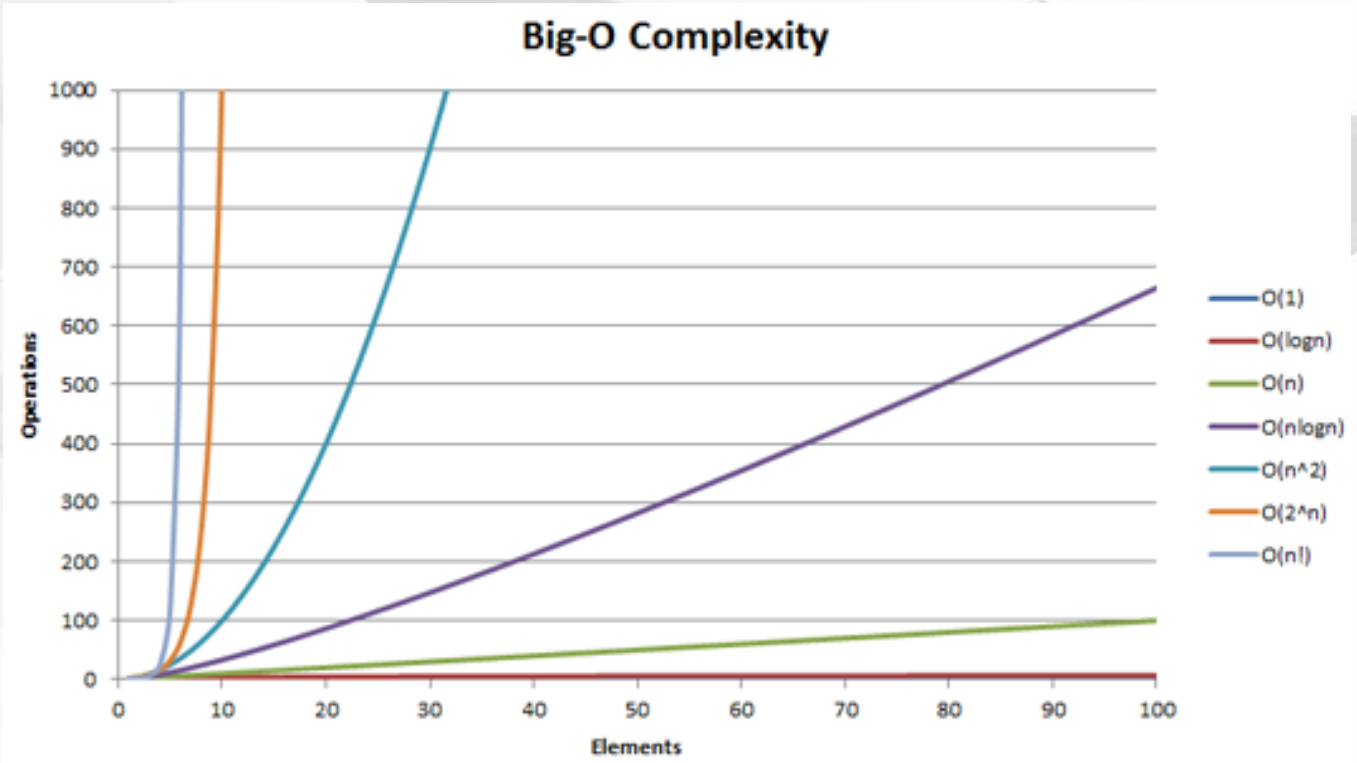
```
typedef struct
{
    int count;
    int stackMax;
    int top1;
    Stacknode *top
}Stack;
```



Complexity of Data Structure

Data structure	Addition	Search	Deletion	Access by index
Array (`T[]`)	O(N)	O(N)	O(N)	O(1)
Linked list (`LinkedList<T>`)	O(1)	O(N)	O(N)	O(N)
Dynamic array (`List<T>`)	O(1)	O(N)	O(N)	O(1)
Stack (`Stack<T>`)	O(1)	-	O(1)	-
Queue (`Queue<T>`)	O(1)	-	O(1)	-
Dictionary, implemented with a hash-table (`Dictionary<K, T>`)	O(1)	O(1)	O(1)	-
Dictionary, implemented with a balanced search tree (`SortedDictionary<K, T>`)	O(log(N))	O(log(N))	O(log(N))	-
Set, implemented with a hash-table (`HashSet<T>`)	O(1)	O(1)	O(1)	-
set, implemented with a balanced search tree (`SortedSet<T>`)	O(log(N))	O(log(N))	O(log(N))	-

Grafik Kompleksitas



- ***Kontrak Perkuliahan***

- Terima Kasih

IPB University
Bogor Indonesia
Department of
Computer Science
http://cs.ipb.ac.id/

Kuliah 2 Struktur Data

Array, Vector, Struct

Disampaikan oleh : Dr. Eng. Annisa, SKom, MKom

Outline

2	Mahasiswa mampu menjelaskan struktur array dan struct, serta mengimplementasikannya	Array dan Struct	Penggunaan array dan struct dalam menyelesaikan masalah: *pattern matching, matrix multiplication*	ANN (K1, K2) HRW (K3)

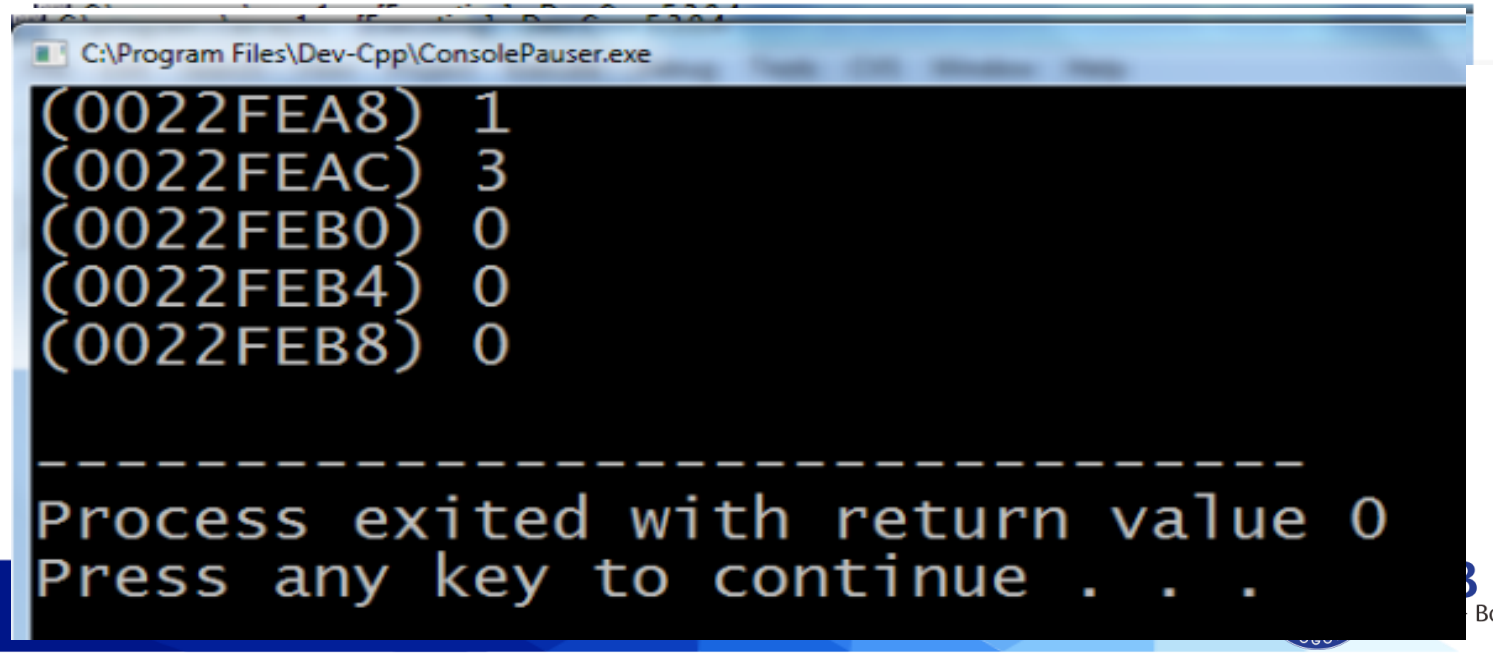
Array

- Struktur data linier yang dapat menyimpan lebih dari satu buah nilai, umumnya bertipe data sama.
- Datanya tersimpan secara terurut di memory.
- Memiliki indeks.
- Cara mendeklarasikannya:

```
typeData nama[kap1][kap2][…]={val1,val2, …}
int A[5]= {1,3,4}
```

Contoh

```
1  #include <stdio.h>
2  int main(){
3      int A[5]={1,3};
4      int k;
5      for(k=0;k<5;k++)printf("(%p)  %d\n",&A[k],A[k]);
6      return 0;
7  }
```



2293416
2293420
2293424
2293428
2293432

Alokasi Dalam Memori

- Alokasi memori dapat dilakukan dengan fungsi:

  `(tipe*)malloc (tsize)`

  `tsize` = ukuran yang diinginkan;

- Dapat memanfaatkan fungsi `sizeof` misalnya `sizeof(int)` akan mengembalikan ukuran yang dibutuhkan oleh sebuah integer.

- Setelah tidak diperlukan, perlu dibebaskan dengan fungsi:

  `free(pointer)`

Implementasi Array

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int n, i;
  int *a, *b;
  scanf("%d", &n);
  a = (int*) malloc (n * sizeof(int));
  b = (int*) malloc (n * sizeof(int));
  for (i = 0; i<n; i++) {
        a[i] = rand()%10;
        b[n-i-1] = a[i];
  }
  for (i = 0; i<n; i++) printf("%d ", a[i]); printf("\n");
  for (i = 0; i<n; i++) printf("%d ", b[i]); printf("\n");
  free(a);
```

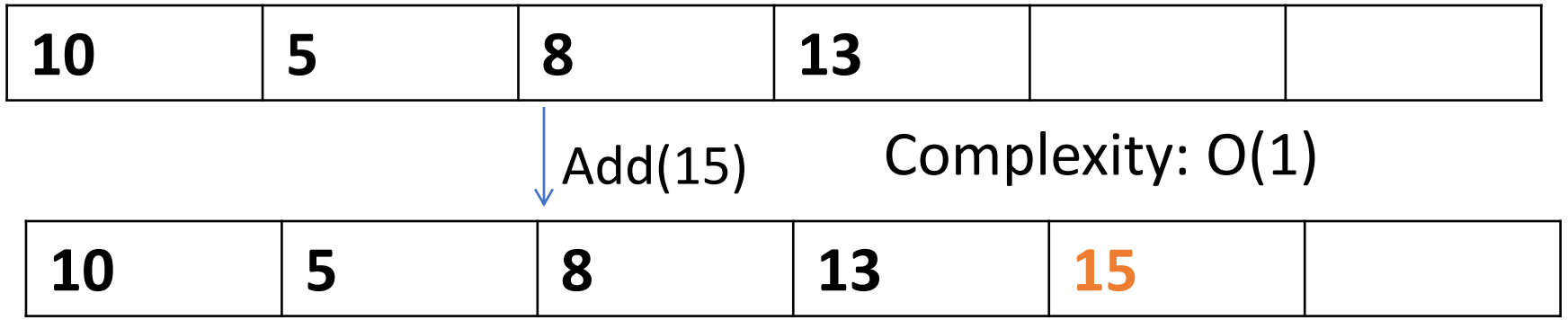
Kelebihan Array

- Waktu akses isi array yang konstan.
- Karena alokasi array terurut dalam memori.
- Contoh array multidimensi:
  - `int arr[3][2]`

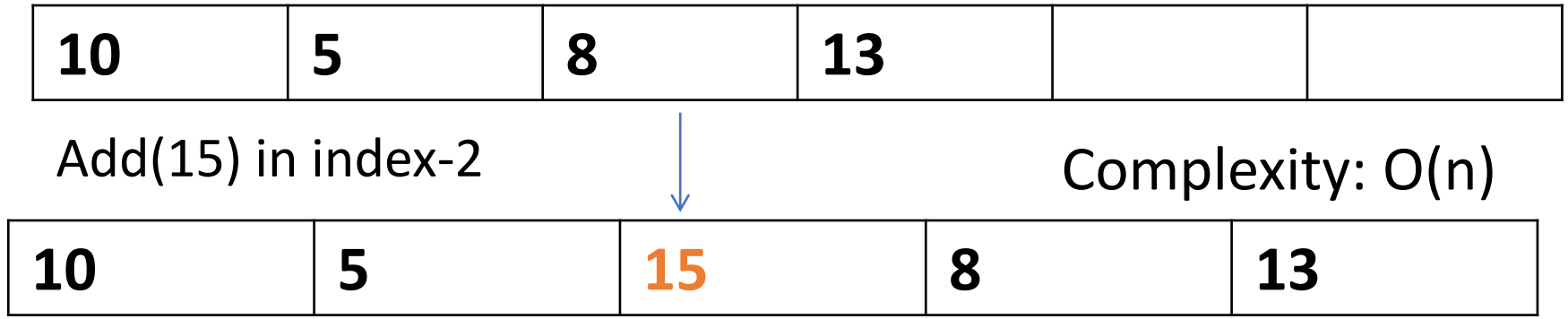
`arr[0][0]`
`arr[0][1]`
`arr[1][0]`
`arr[1][1]`
`:`
`arr[2][1]`

Array Operation (Add)

- Position: End



- Position: Middle



Problems with Static Array

- The process of allocating memory at compile time is known as **static memory** allocation
- The arrays that receives static memory allocation are called **static arrays.**
- Declaring an array with a fixed size like int a[100000]; has some typical problems:
  - The programmer has no control over the size of the data sets the user is interested in.
  - Erroneous assumptions that a maximum will never be exceeded are the source of many programming bugs.
  - Declaring very large arrays can be extremely wasteful of memory.
  - Fixed size arrays can not expand as needed.

Dynamic Array

- It is possible to allocate memory to arrays at run time. This feature is known as **dynamic memory** allocation.
- The arrays created at run time are called **dynamic arrays**.
- Dynamic arrays dynamically allocating an array of the right size, or reallocating an array when it needs to expand.
- Both of these are done by declaring an array as a pointer and using the **new** operator to allocate memory, and **delete** to free memory that is no longer needed.

Static Array & Dynamic Array

```
const int MAX = 1000;
int a[MAX];
int n = 0;

//--- Read into the array
while (cin >> a[n]) {
    n++;
}

//--- Write out the array
for (int i=0; i<n; i++) {
    cout << a[i] << endl;
}
```

This program fragment will break if more than 1000 integers are entered.

```
int max = 10;            // no longer const
int* a = new int[max];
int n = 0;

//--- Read into the array
while (cin >> a[n]) {
    n++;
    if (n >= max) {
        max = max * 2;              // double the previous size
        int* temp = new int[max]; // create new bigger array.
        for (int i=0; i<n; i++) {
            temp[i] = a[i];         // copy values to new array.
        }
        delete [] a;                // free old array memory.
        a = temp;                   // now a points to new array.
    }
}
```

This program is only limited by the amount of memory. It starts with a small array and expands it only if necessary.

Vector in C++ STL

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
- **STL components :**
  - A Container is a way that stored data is organized in memory
  - Algorithms are procedures that are applied to containers
  - Iterators are generalization of the concept of pointers, they points to elements in a container

Dynamic Array & Vector

- Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- Dynamic Arrays **have to be *deallocated explicitly*** whereas vectors are ***automatically de-allocated*** .

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int* arr = new int[100];
    delete[] arr;

    vector<int> v;
    return 0;
}
```

Dynamic Array & Vector

- **Size of arrays are *fixed*** whereas the **vectors are *resizable***

```
Size of Array 100
Size of vector Before Removal=5
1 2 3 4 5
Size of vector After removal=4
1 2 4 5
```

Define container, algorithm, and iterator !

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
        int array[100]; // Static Implementation
        cout << "Size of Array " << sizeof(array) / sizeof(array[0]) << "\n";
        vector<int> v; // Vector's Implementation
        // Inserting Values in Vector
        v.push_back(1);
        v.push_back(2);
        v.push_back(3);
        v.push_back(4);
        v.push_back(5);
        cout << "Size of vector Before Removal=" << v.size() << "\n";
        // Output Values of vector
        for (auto it : v)
                cout << it << " ";
        v.erase(v.begin() + 2); // Remove 3rd element
        cout << "\nSize of vector After removal=" << v.size() << "\n";
        // Output Values of vector
        for (auto it : v)
                cout << it << " ";
        return 0;
}
```

Contoh Kasus 1

- Diketahui beberapa nilai bilangan bulat (1 s/d 1000) sebanyak maksimum 10^7 bilangan. Selanjutnya, Anda diminta membuat program untuk mencari suatu nilai bilangan apakah ada ataukah tidak ada di dalam array tersebut.
- Contoh input
  - 5 9 8 7 6 8 7 6 2 3 4 5 1 …
  - 7 9 10
- Contoh output:
  - 7 ada
  - 9 ada
  - 10 tidak ada

Solusi Kasus 1

- Alternatif 1
  - Baca semua data dan masukkan ke dalam array
    - Sort
    - Search
- Alternatif 2 (optimum)
  - Karena nilai data bulat 1-1000, maka buat indeks array sebanyak 1000 (0-999)
  - Baca data, dan isi nilai 1 ke array dengan indeks=data-1

Contoh Kasus 2

- Diketahui beberapa nilai bilangan riil sebanyak maksimum 10^7 bilangan. Selanjutnya, Anda diminta membuat program untuk mencari suatu nilai bilangan apakah ada ataukah tidak ada di dalam array tersebut.
- Contoh input
  - 5.5 9.0 8.0 7.0 6.0 8.0 7.0 6.0 2.0 3.0 4.0 5.0 1.0 …
  - 5.5 9.0 10.0 3.0
- Contoh output:
  - 5.5 ada
  - 9.0 ada
  - 10.0 tidak ada
  - 3.0 ada

Solusi Kasus 2

- Cara seperti kasus 1 tidak dapat dilakukan karena nilai data adalah bilangan riil sementara indeks array adalah bilangan bulat.
- Jika semua data disimpan ke dalam array, maka melebihi kapasitas array.
- Membutuhkan struktur data array yang diimplementasikan dalam bentuk vektor, tidak membutuhkan inisalisasi kapasitas seperti array.

Vector

- Provides an alternative to the built in array.
- A vector is self grown.
- Use It instead of the built in array!

Defining a new vector

Syntax: vector<of what>

For example :

- vector<int> - vector of integers.
- vector<string> - vector of strings.
- vector<int * > - vector of pointers to integers.
- vector<Shape> - vector of Shape objects, where Shape is a user defined class.

Using Vector

- #include <vector>

- Two ways to use the vector type:
  1. Array style.
  2. STL style

Using a Vector – Array Style

```
void simple_example()
{
        const int N = 10;
        vector<int> ivec(N);

        for (int i=0; i < 10; ++i)

                cin >> ivec[i];


        int ia[N];
        for ( int j = 0; j < N; ++j)
                ia[j] = ivec[j];
}
```

Meskipun dideklarasikan seperti array, namun tetap diperlakukan seperti vektor

Using a vector – STL style

define an empty vector

```
vector<string> svec;
```

insert elements into the vector using the method push_back.

```
string word;
while ( cin >> word ) //the number of words is unlimited.
{
        svec.push_back(word);
}
```

Insertion

Inserts an element with value x at the end of the controlled sequence.

```
svec.push_back(str);
```

Size

unsigned int size();

Returns the length of the controlled sequence (how many items it contains).

```
unsigned int size = svec.size();
```

Class Exercise 1

Write a program that read integers from the user, sorts them, and print the result.

Solving the problem

- Easy way to read input.
- A “place” to store the input
- A way to sort the stored input.

Using STL

```
int main()
{
        int input;
        vector<int> ivec;

        /* rest of code */
}
```

STL - Input

```
while ( cin >> input )
        ivec.push_back(input);
```

STL - Sorting

```
sort (ivec.begin(), ivec.end());
```

Sort Prototype:

```
void  sort(Iterator first, Iterator  last);
```

STL - Output

```
for ( int i = 0; i < ivec.size(); ++i )
        cout << ivec[i] << " ";
cout << endl;
```

```
vector<int>::iterator it;
for ( it = ivec.begin(); it != ivec.end(); ++it )
        cout << *it << " ";
cout << endl;
```

STL - Include files

```
#include <iostream>  // I/O
#include <vector>    // container
#include <algorithm> // sorting

//using namespace std;
```

Putting it all together

```
int main() {
      int input;
      vector<int> ivec;

      // input
      while (cin >> input )
            ivec.push_back(input);

      // sorting
      sort(ivec.begin(), ivec.end());

      // output
      vector<int>::iterator it;
      for ( it = ivec.begin();
            it != ivec.end(); ++it ) {
                  cout << *it << " ";
      }
      cout << endl;

      return 0;
}
```

Operations on vector

- iterator begin();
- iterator end();
- bool empty();
- iterator erase(iterator it);
- iterator erase(iterator first, iterator last);
- void clear();
- …

Struct

- Suatu struktur data dari serangkaian nilai yang dimungkinkan berbeda tipe datanya
- Cara mendeklarasikannya:

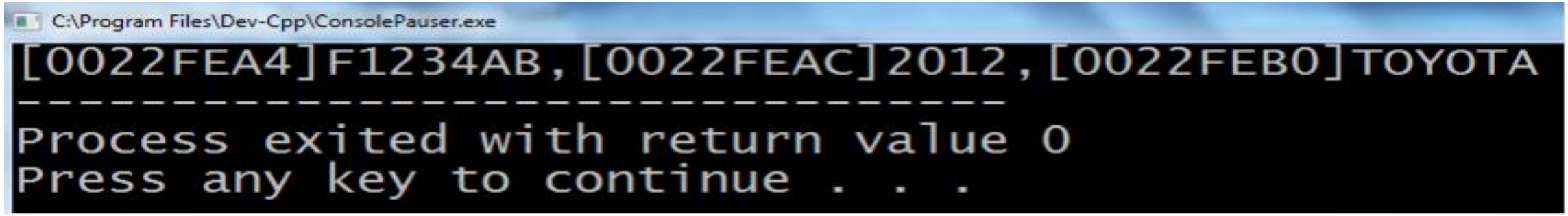
```
struct namaStruct {
  tipedata1 elemen1;
  tipedata2 elemen2;
  …
  tipedataN elemenN;
};
```

Contoh Struct

```c
#include <stdio.h>
struct mobil{
    char platNo[8];
    int tahun;
    char merk[16];
};

int main(){
    struct mobil S1={"F1234AB",2012,"TOYOTA"};
    printf("[%p]%s,[%p]%d,[%p]%s",&S1.platNo,S1.platNo,
                &S1.tahun,S1.tahun,&S1.merk,S1.merk);

    return 0;
}
```

Contoh Struct (2)

2293412 2293420 2293424



- Setiap elemen data akan dialokasikan terurut meskipun masing-masing memiliki ukuran yang berbeda

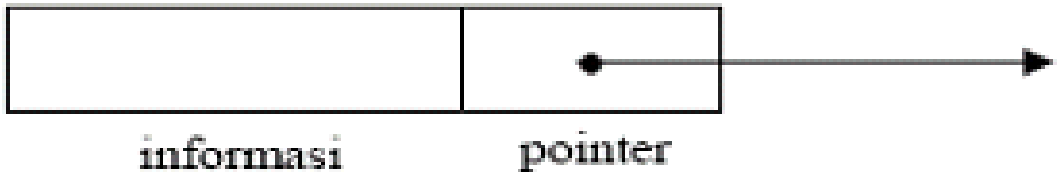
Struktur Data Linear

Linked List

- Daftar yang saling berkait
- Elemennya sering disebut node
- Node-node pembentuk LL tidak harus disimpan secara terurut di memori → tidak memiliki indeks
- Sekuensial akses
- Node tersusun atas dua komponen data: informasi dan pointer

Node

- Abstraksi Logik

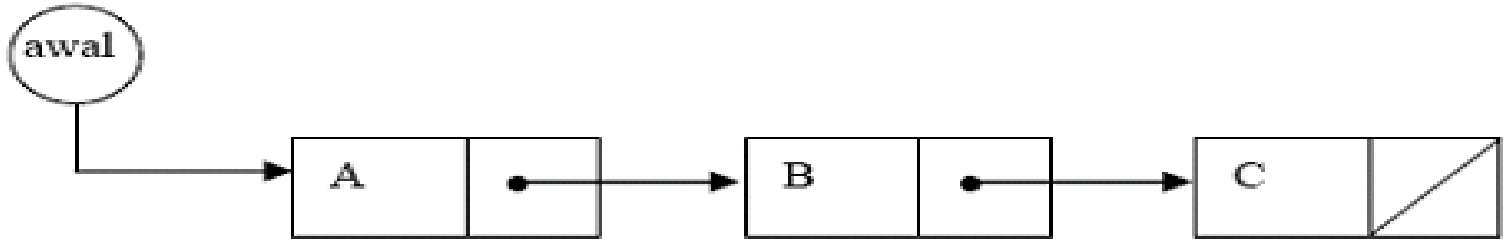


- Kode

```
struct node{
    komponen informasi;
    komponen pointer;
}
```

Contoh Linked List

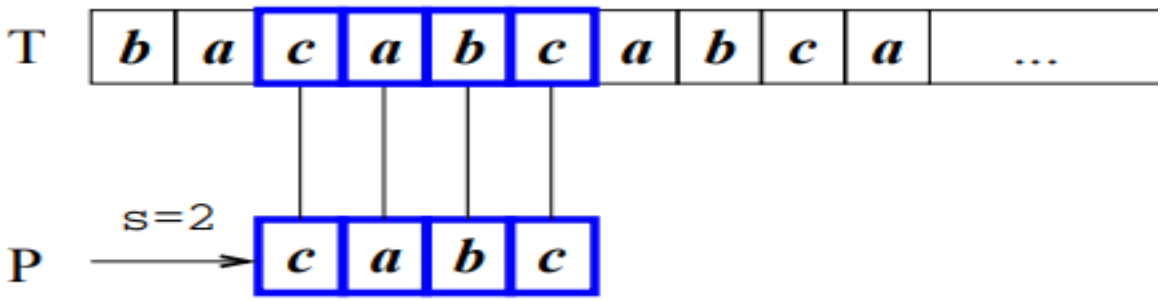
- Linked list dengan 3 buah node



- https://www.w3resource.com/c-programming-exercises/array/index.php

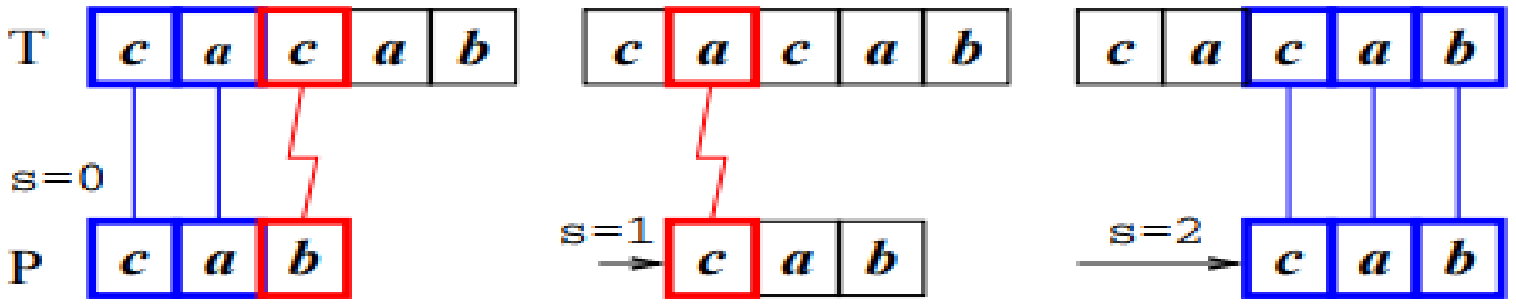
Pelajari : String Matching Problem (SMP)

- Diberikan sebuah array of char $T[1…n]$ dan sebuah pola $P[1…m]$ sehingga elemen dari T dan P merupakan anggota dari $\sum = \{0,10\}$ atau $\sum = \{a,b,c,\ldots,z\}$
- Pola P dideteksi dengan menggeser s karakter di dalam T, SMP mencari semua kemungkinan nilai s, di mana $0 \leq s \leq n-m$



Brute-force Algorithm (SMP)

- Pertama-tama, P disejajarkan dengan index awal T. P kemudian dibandingkan dengan T dari kiri-kanan.
- Jika mismatch, geser P ke kanan sebanyak 1 dan mulai kembali pembandingan.



```c
#include<stdio.h>
#include<string.h>
void search(char *pat, char *txt)
{
int M = strlen(pat);
int N = strlen(txt);

/* A loop to slide pat[] one by one */
for (int i = 0; i <= N - M; i++)
{
int j;

/* For current index i, check for pattern match */
for (j = 0; j < M; j++)
{
if (txt[i + j] != pat[j])
break;
}
if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
{
printf("Pattern found at index %d \n", i);
}
}
}

/* Driver program to test above function */
int main()
{
char *txt = "AABAACAADAABAAABAA";
char *pat = "AABA";
search(pat, txt);
return 0;
}
```

Diskusi Kelompok

- Diketahui T:

a	b	a	b	a	b	a	b	c

dan P:

a	b	a	b	a	b	c

- Apa keluarannya?,
- Berapa kompleksitasnya?,

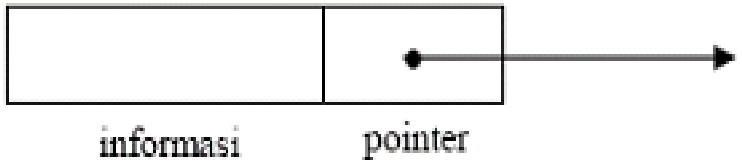
- Terima Kasih

LINKED LIST

Pertemuan 3 Kuliah Struktur Data

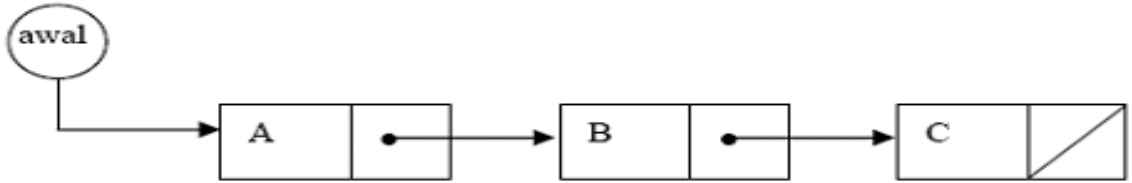
Linked List

- **Untaian** atau **rangkaian** berantai dari node-node yang disusun dengan bantuan **pointer**.
- Masing-masing **node** merupakan suatu **record** yang berisi dua bagian, yaitu bagian **informasi** dan bagian **pointer** (yang berfungsi untuk menunjuk node berikutnya).
- Antara node yang satu dengan node berikutnya pada linked list **tidak harus disimpan secara terurut dalam memori (tidak memiliki indeks).**



Contoh Linked List

- Linked list dengan 3 buah node



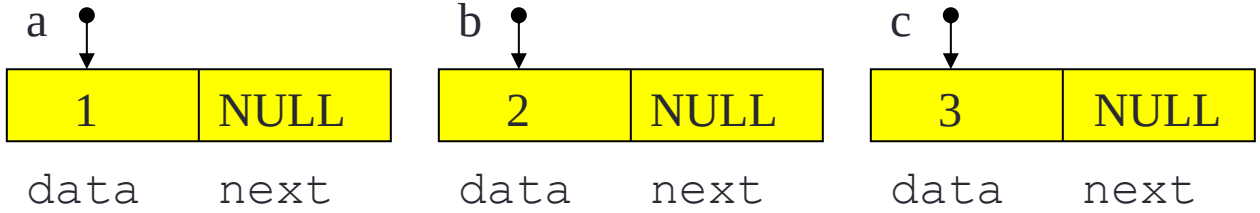
Deklarasi Node Pada Linked List

```
struct node{
    int data;
    struct node *next;
};
typedef struct node node;
```

Variabel pointer next disebut *Link* berisikan sebuah alamat yang dialokasikan di memori sebagai *successor list*

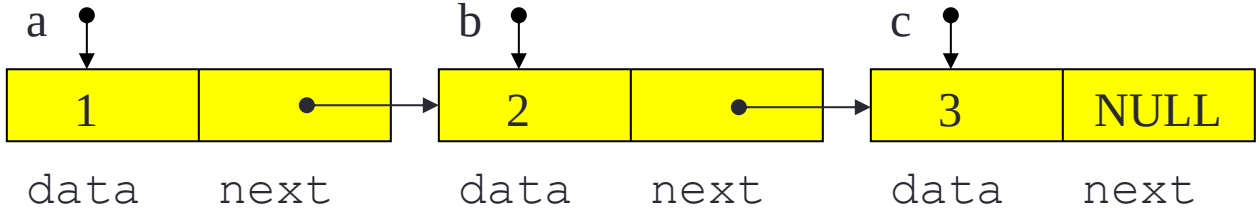
Contoh

```
node *a, *b, *c;
a = (node*)malloc(sizeof(node));
b = (node*)malloc(sizeof(node));
c = (node*)malloc(sizeof(node));

a->data = 1; a->next = NULL;
b->data = 2; b->next = NULL;
c->data = 3; c->next = NULL;
```



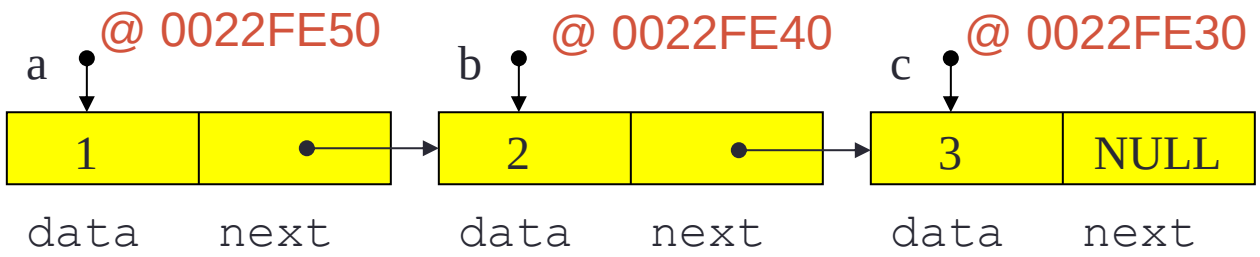
Contoh

```
node *a, *b, *c;
a = (node*)malloc(sizeof(node));
b = (node*)malloc(sizeof(node));
c = (node*)malloc(sizeof(node));

a->data = 1; a->next = b;
b->data = 2; b->next = c;
c->data = 3; c->next = NULL;
```



```
a->data      = 1
b->data      = 2
c->data      = 3
a->next      = 0022FE40
b->next      = 0022FE30
c->next      = NULL
```

```
a->next->data      => 2
b->next->data      => 3
c->next->data      => ERROR!
a->next->next      => 0022FE30
b->next->next      => NULL
c->next->next      => ERROR!
```



Bentuk Operasi Pada Linked List

- Penambahan data
- Penghapusan data
- Pengaksesan data (menampilkan 1 atau seluruh linked list, termasuk pencarian)
- Update data

SINGLE LINKED LIST
PENAMBAHAN DATA

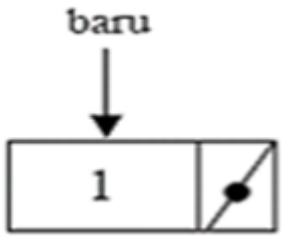
Penambahan Data (*inserting*)

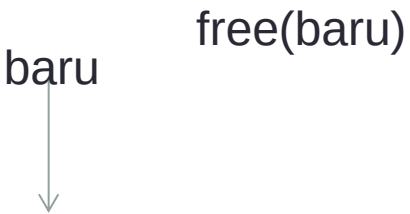
- Operasi penambahan data baru pada linked list dapat memiliki 3 mode
  - **Di awal list**
  - **Di antara data yang lain**
  - **Di akhir list**

Penambahan Data (*inserting*)

- Node yang akan dimasukkan harus dialokasikan terlebih dahulu pada memory

```
baru=malloc(sizeof(node));
```





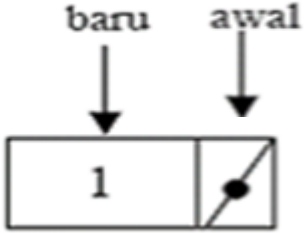
Penambahan Data di Awal

- Ketika sebuah **data (node) baru** akan dimasukan pada linked list **kondisi linked list bisa kosong ataupun sudah ada isinya**
- **Node pertama** pada sebuah linked list pada umumnya ditunjuk sebagai suatu pointer **head/awal**
- Kondisi kosong dapat dicek dengan **kondisi awal=NULL**

Penambahan Data di Awal (2)

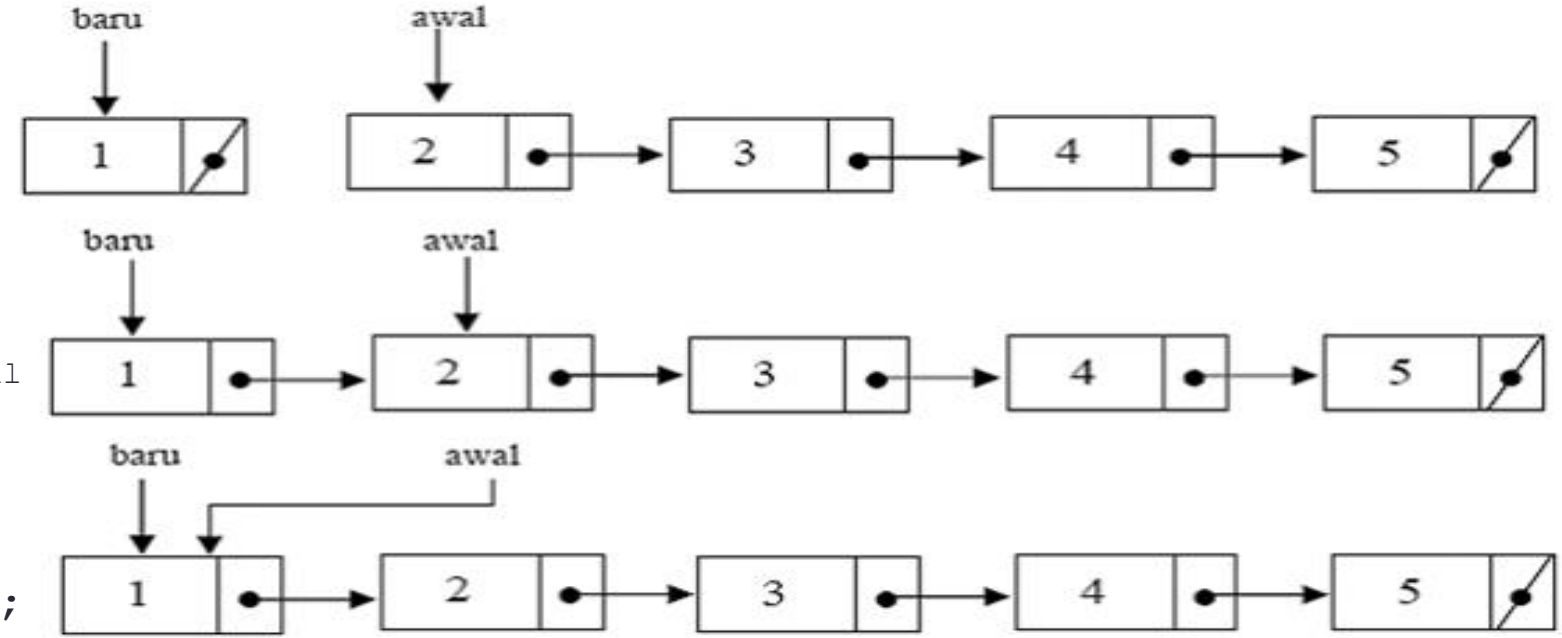
- Pada kondisi kosong penambahan dapat dilakukan dengan cara menjadikan **node baru** tersebut sebagai node yang ditunjuk oleh awal.

```
awal=baru;
```



- Pada saat sudah ada isinya penambahan memiliki beberapa **tahapan/Prosedur**:
  (1) Node baru->next menunjuk ke node awal
  (2) Node awal menunjuk ke node baru

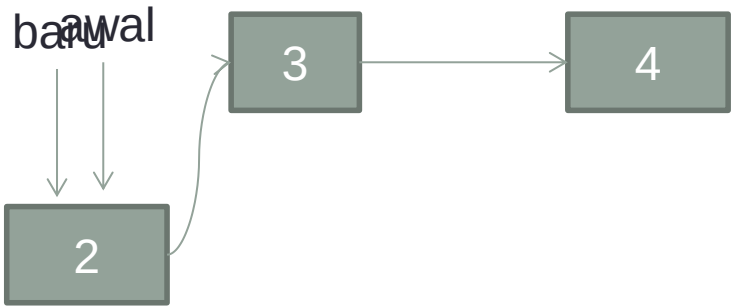
Penambahan Data di Awal (3)



Penambahan Data di Awal (4)

- Sintax dalam C (tentunya dengan inisalisasi terlebih dahulu)

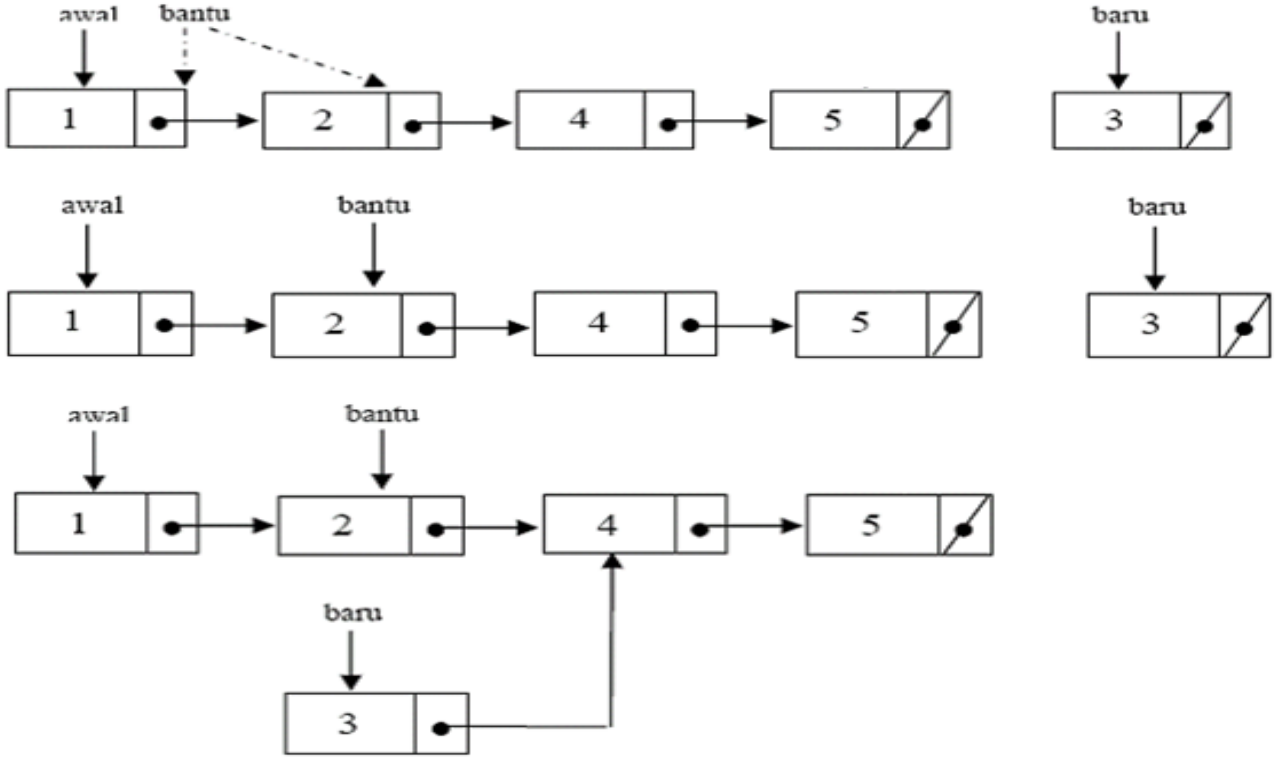
```
node* awal
baru=malloc(sizeof(node));
baru->data=dataBaru;
if (awal==NULL) {
    awal=baru;
    baru->next=NULL;
} else {
    baru->next=awal;
    awal=baru;
}
```



Penambahan Data di antara List

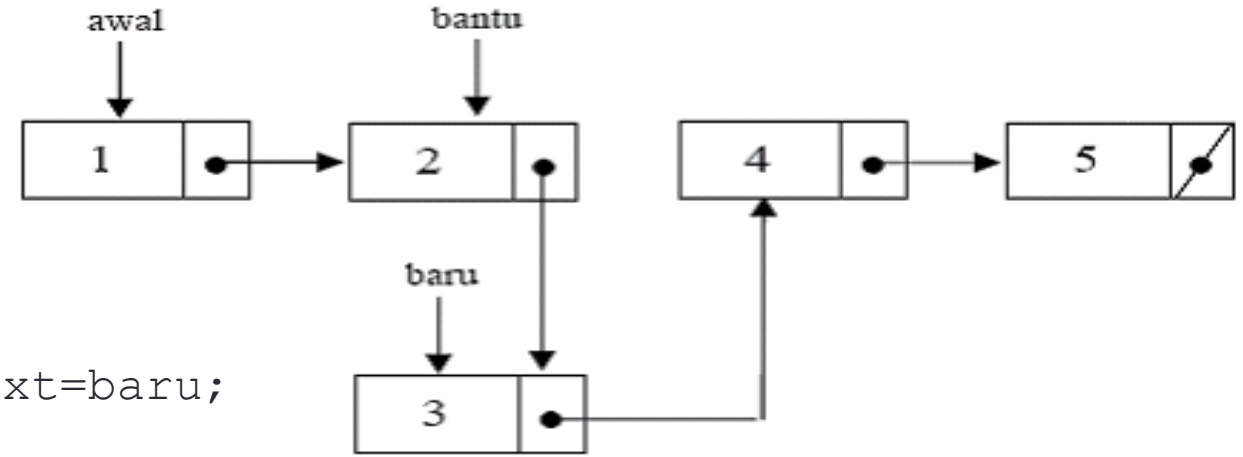
bantu=awal

```
while(bantu->next->data!=4)
        bantu=bantu->next;
```




```
baru->next=bantu->next;
```

Penambahan Data di antara List (2)




```
bantu->next=baru;
```

Penambahan Data di antara List (3)

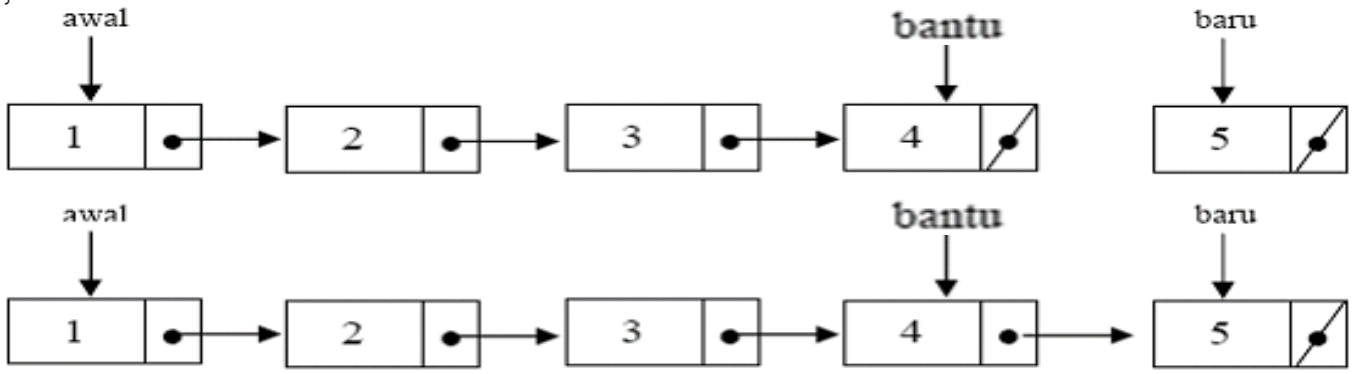
Bagaimana Algoritme dan sintaxnya di dalam C?

```
baru=malloc(sizeof(node));
baru->data=dataBaru;
if (awal==NULL){
      awal=baru;
      baru->next=NULL;
  } else {
     bantu = awal;
      while(bantu->next->data!=4)
                bantu=bantu->next;
      baru->next=bantu->next;
      bantu->next=baru;
}
```

Penambahan di Akhir List

Temukan node terakhir (nilai next == NULL) → node terakhir dari list

```
while (bantu->next != NULL) {
      bantu = bantu->next;
}
```




```
bantu->next = baru
```

Penambahan di Akhir List (2)

- **Bisa dengan menambahkan pointer akhir, sehingga tidak perlu penelusuran setiap kali melakukan insert diakhir list**

- **Bagaimana Prosedurnya?**

- Pointer “akhir” pertama kali menunjuk ke “awal”
- Gerakkan terus pointer “akhir” setiap kali ada pemasukan data diakhir.

- **Cara gerakkan pointer sampai terakhir?**

```
while (bantu->next != NULL) {
        bantu = bantu->next;
}
```

SINGLE LINKED LIST
PENGHAPUSAN DATA

Penghapusan Data (*deleting*)

- Operasi penghapusan data baru pada linked list dapat memiliki 3 mode
  - **Di awal list**
  - **Di antara data yang lain**
  - **Di akhir list**

Penghapusan Data di Awal List

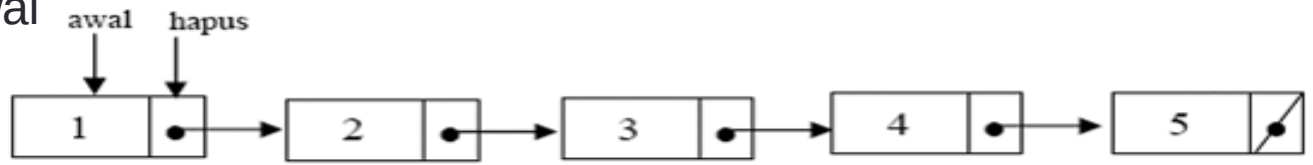
Prosedurnya?

- Tentukan node yang ingin dihapus sebagai node “hapus”
- Gerakkan node awal satu langkah ke setelahnya

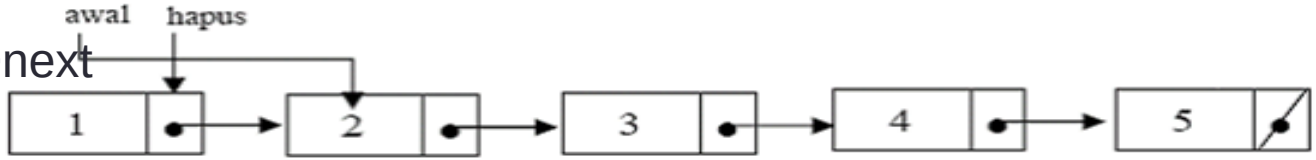
```
hapus=awal;
awal=awal->next;
free(hapus);
```

Penghapusan Data (*deleting*)

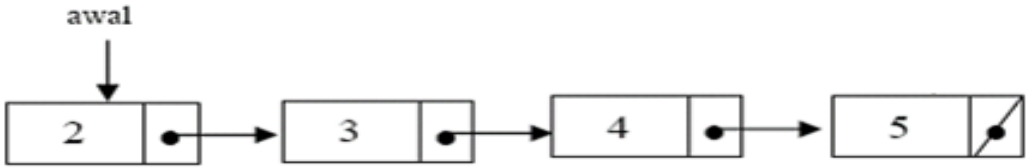
hapus=awal



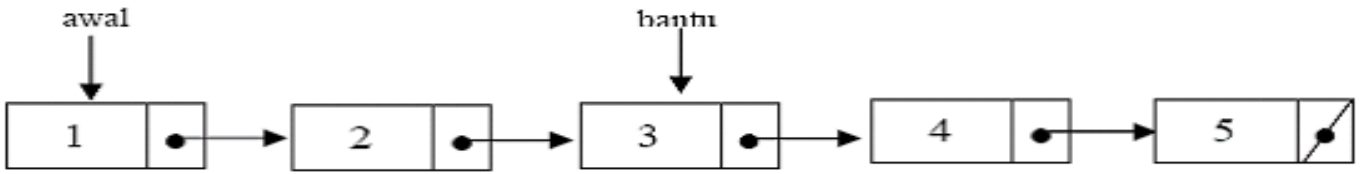
awal=awal->next



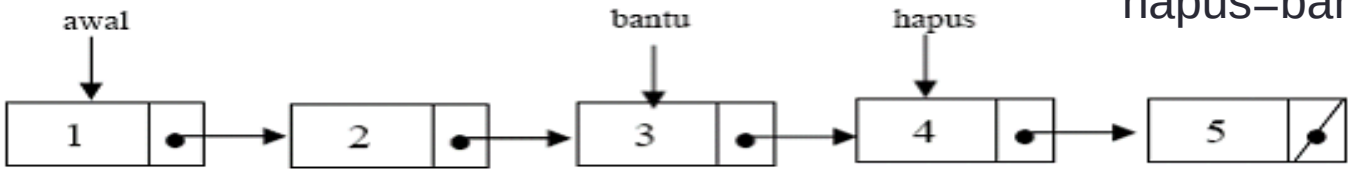
Free(hapus)



Penghapusan List Node Tertentu



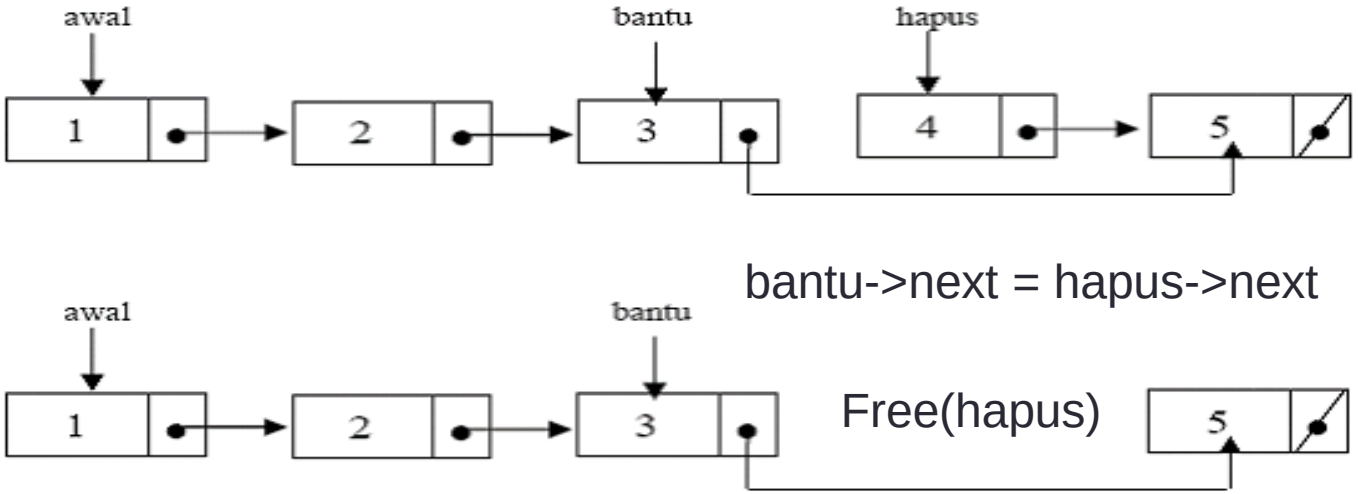
hapus=bantu->next



Ilustrasi (penghapusan node dengan data=4)

Pointer “bantu” menunjuk node sebelum node yang ingin dihapus

Penghapusan List Node Tertentu



Ilustrasi (penghapusan node dengan data=4)

Pointer “bantu” menunjuk node sebelum node yang ingin dihapus

SINGLE LINKED LIST
PENGAKSESAN & UPDATE DATA

Pengaksesan Data

- Tidak semudah array yang dapat menggunakan *index* untuk menuju suatu data tertentu
- Suatu node pada linked list dengan pointer tunggal hanya memiliki info lokasi data (node) next
- Penelusuran secara sekuensial (linier) dari awal sampai dengan posisi ataupun node (tujuan) dengan karakteristik data tertentu
- Kasus terbaik 1 (di awal), terburuk n (di akhir atau tidak ditemukan)

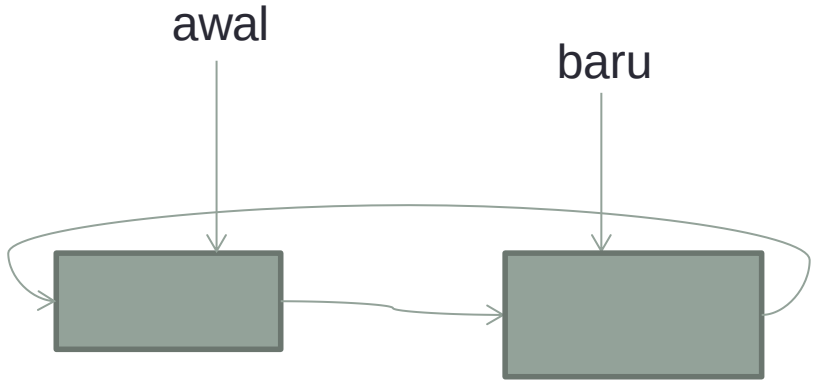
Update Data

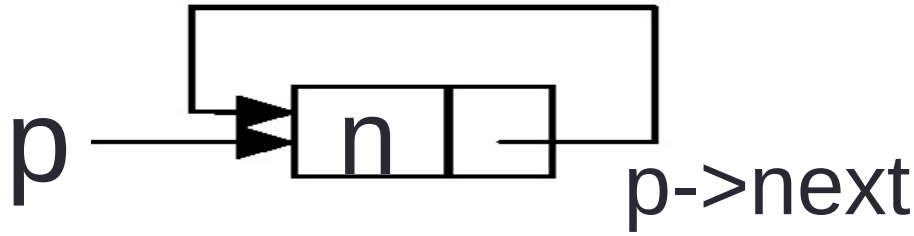
- Pada dasarnya update data akan diawali penelusuran data yang ingin diubah atau proses mengakses data yang dituju
- Ketika posisi data telah ditemukan secara langsung data terkait pada node dapat langsung diubah dengan data yang baru

CIRCULAR **LINKE**D LIST

Circular Linked List (CLL)

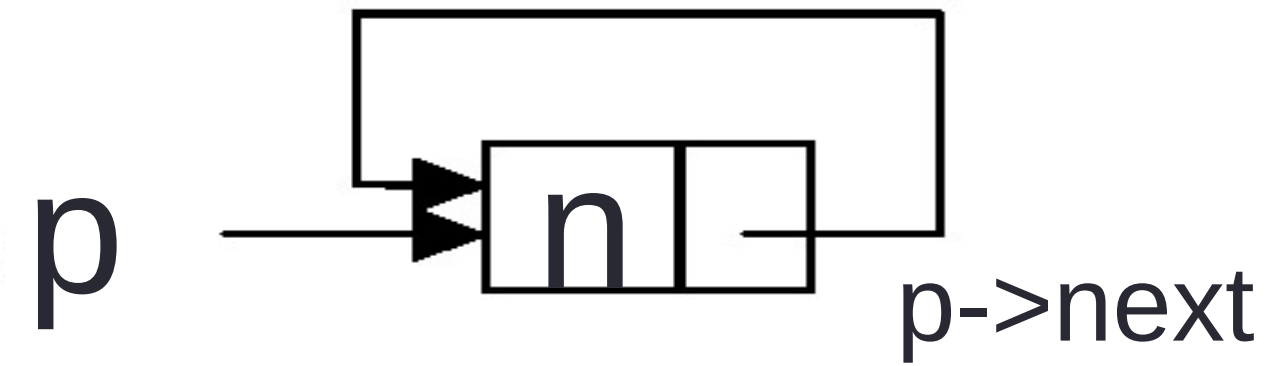
- Suatu LL yang pada node terakhirnya menunjuk ke alamat node awal.
- Penelusuran satu arah dan circular





awal->next =baru
Baru->next = awal

Tambah Node baru

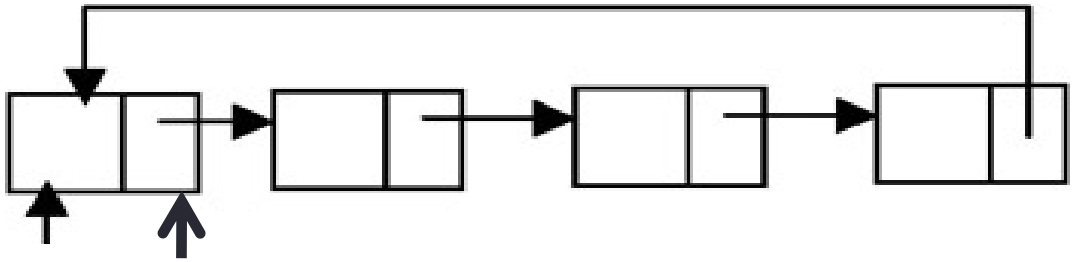


p-> data = n;

p-> next = p;

Tambah Node di Akhir

- Mencari node paling akhir, baru ditambahkan

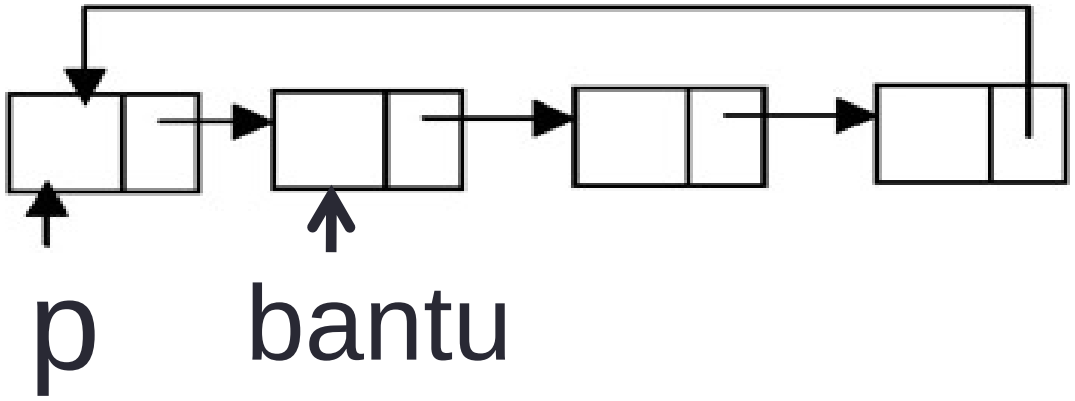


p bantu

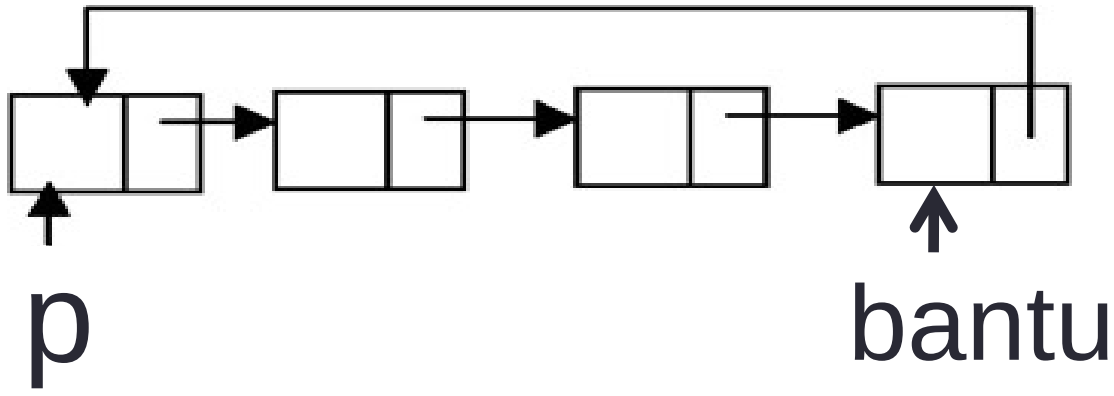
```
bantu = p;
```

Tambah Node di Akhir

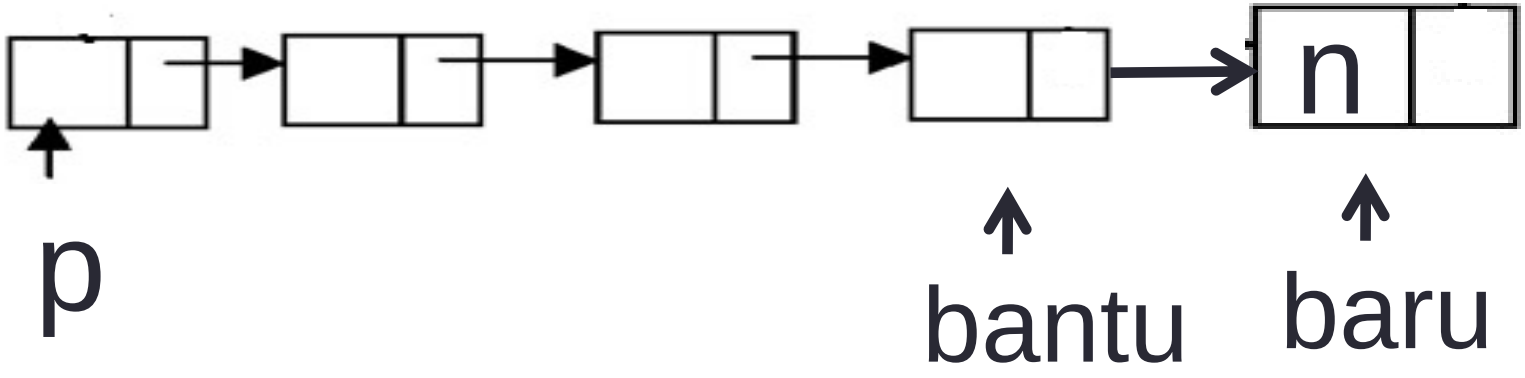
```
while (bantu->next != p)
        bantu = bantu-> next;
```



Tambah Node di Akhir

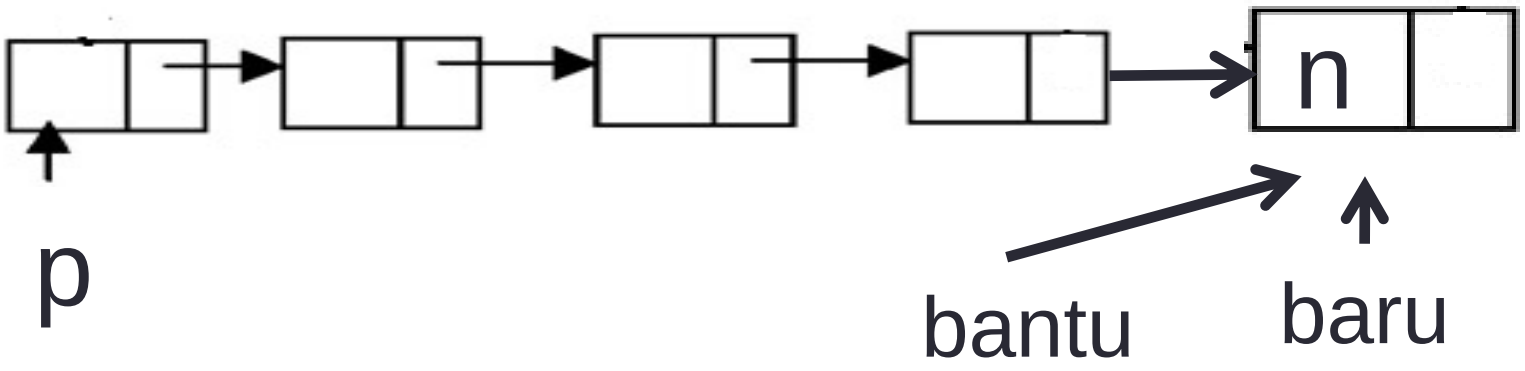


Tambah Node di Akhir



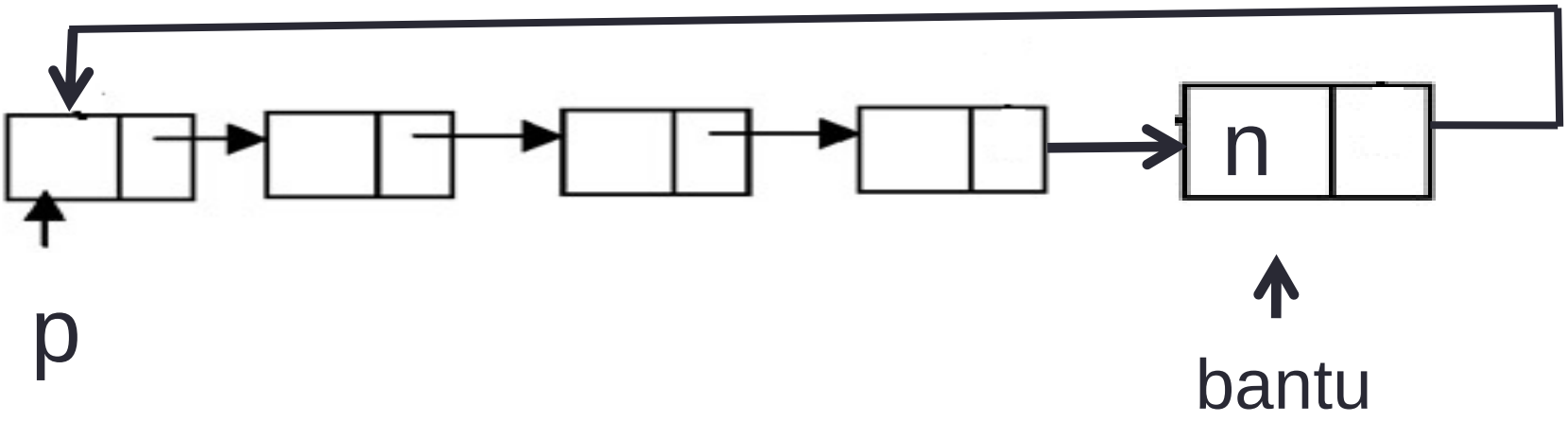

```
bantu->next  = baru;
baru->data = n;
```

Tambah Node di Akhir




```
bantu = baru;
```

Tambah Node di Akhir

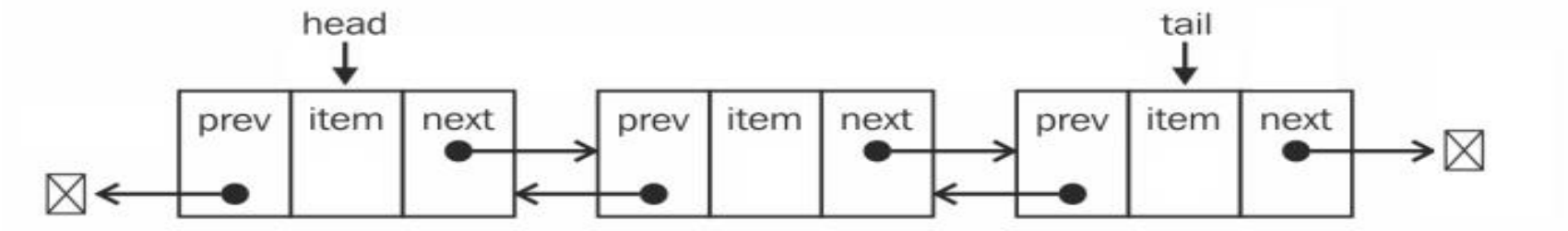



```
bantu->next = p
```

DOUBLY LINKED LIST

Doubly Linked List

- Suatu LL yang tiap nodenya memiliki 2 buah pointer penunjuk (prev & next / left & right)
- Setiap node akan memiliki info posisi node sebelumnya dan sesudahnya
- List “dipegang” oleh dua pointer, yaitu ujung awal/head dan ujung akhir/tail

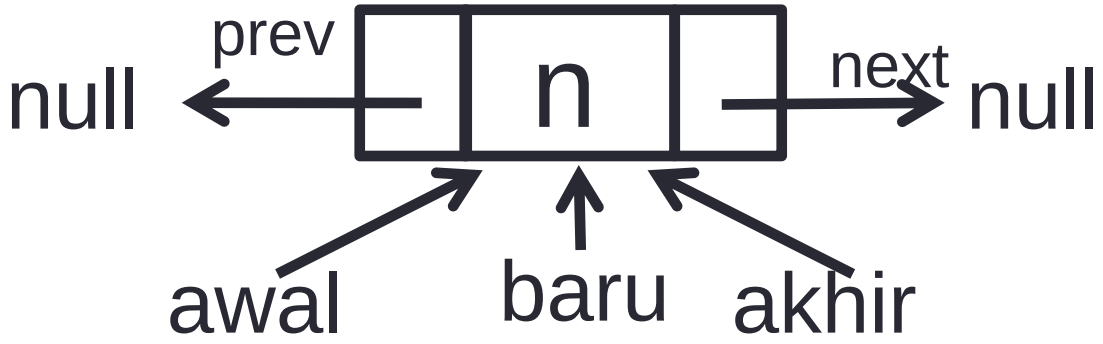


Deklarasi Struct pada DLL

```
struct node {
        int data;
        struct node *prev, *next;
};
```

Tambah Node Baru

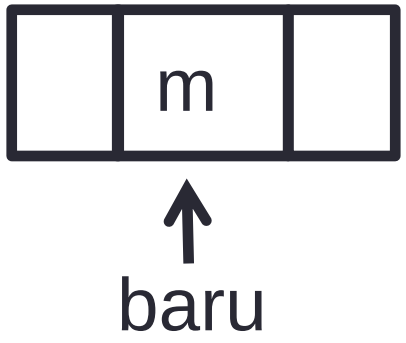
- Jika List masih kosong

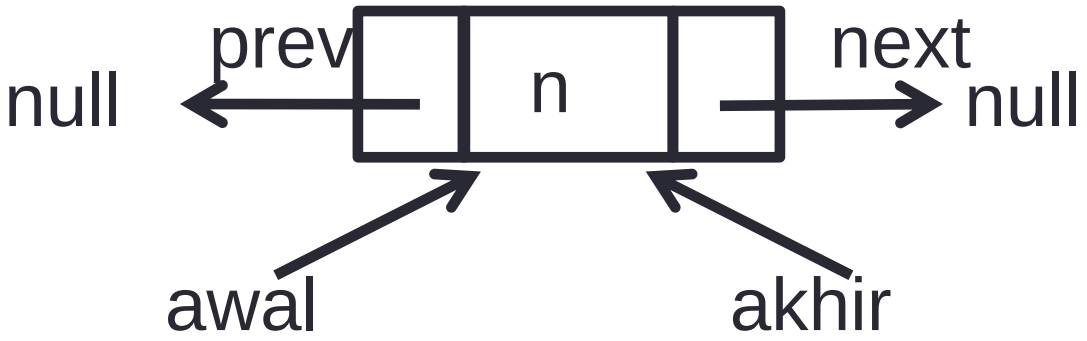



```
baru->data = n;
baru->prev = baru->next = NULL;
awal = akhir = baru;
```

Tambah Node Baru di Awal

- Jika list sudah terisi sebelumnya

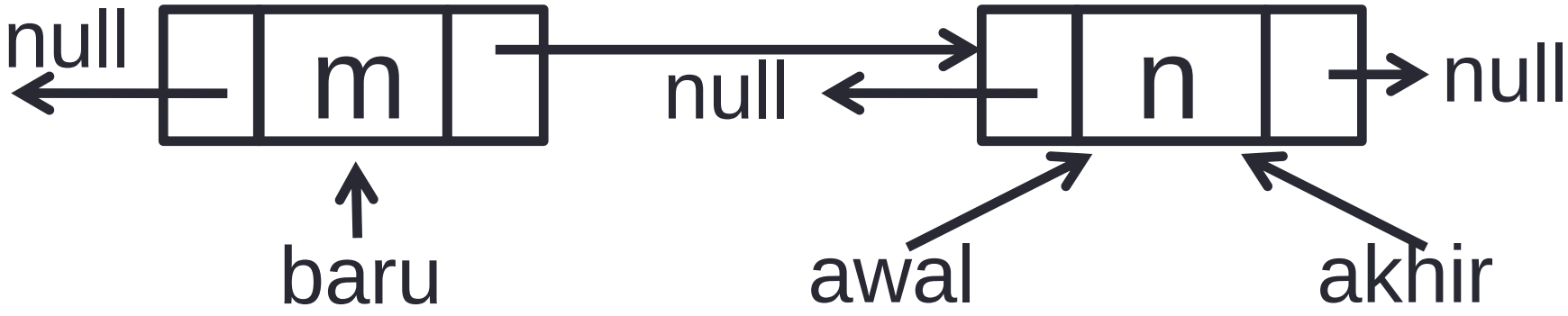





```
node *baru;
baru=(node*)malloc(sizeof(node));
baru->data=m;
```

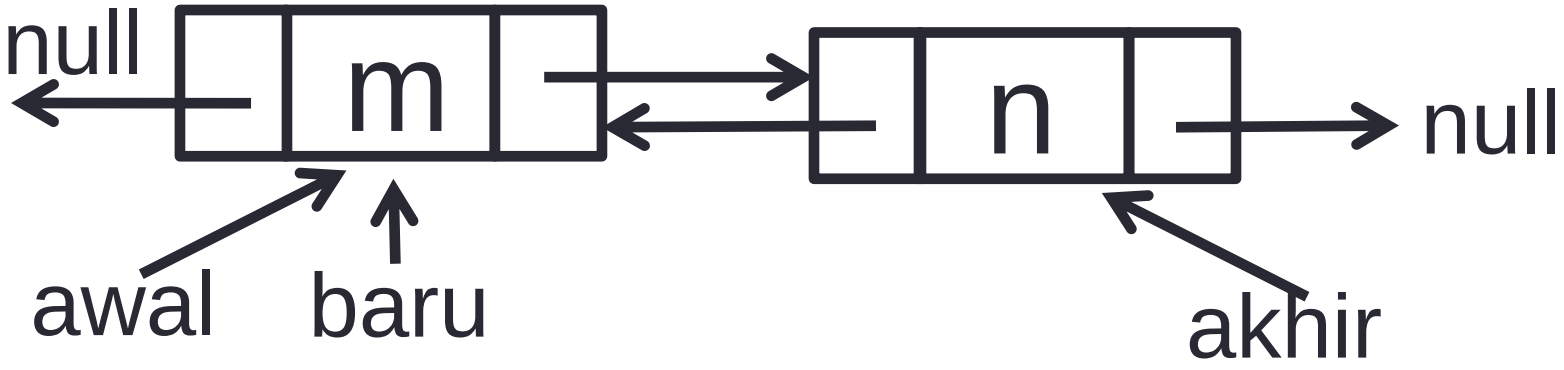
Tambah Node Baru di Awal

- Jika List sudah terisi node




```
baru->next=awal;
```

Tambah Node Baru di Awal




```
awal->prev=baru;
awal=baru;
```

	Array	Linked List
Cost of accessing elements	O(1)	O(n)
Insert/Remove from beginning	O(n)	O(1)
Insert/Remove from end	O(1)	O(n)
Insert/Remove from middle	O(n)	O(n)

Arrays Vs Linked Lists

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Diskusi Kelompok

Diskusikan secara berkelompok operasi:

- Penambahan (awal, tengah, akhir)
- Penghapusan (awal, tengah, akhir)

Diskusi 1

- Given a singly linked list, Your task is to remove every K-th node of the linked list. Assume that K is always less than or equal to length of Linked List.

Diskusi 2

Misalkan node dari sebuah linked list diimplementasikan sebagai berikut:

```
struct Node
{
int data;
struct Node *next;
};
```

Jika diasumsikan list sudah terbentuk 1->2->3->4->5->6, apakah yang dilakukan fungsi fun di bawah ini terhadap linked list tersebut?

```
void fun(struct Node* head)
{
if(head== NULL)
        return;
printf("%d ", head->data);

if(head->next != NULL )
        fun(head->next->next);
printf("%d ", head->data);
}
```

Diskusi 3

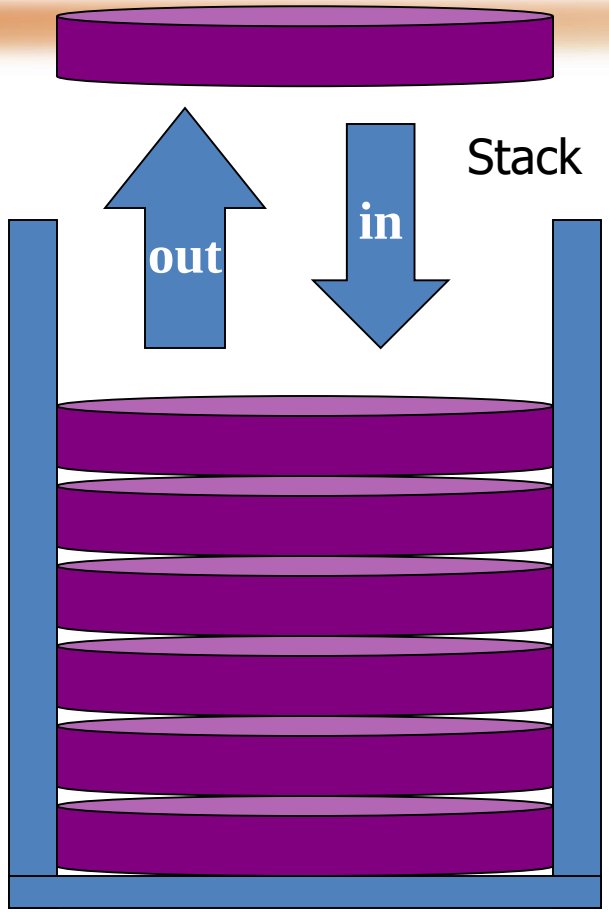
	Array	Linked List
Size		
Insertion & Deletion		
Random Access		
Memory Allocation		
Sequential Access		
BigO for Cost of accessing element		
BigO for insert/remove from beginning position		
BigO for insert/remove from end position		
BigO for insert/remove from middle position		

TERIMA KASIH

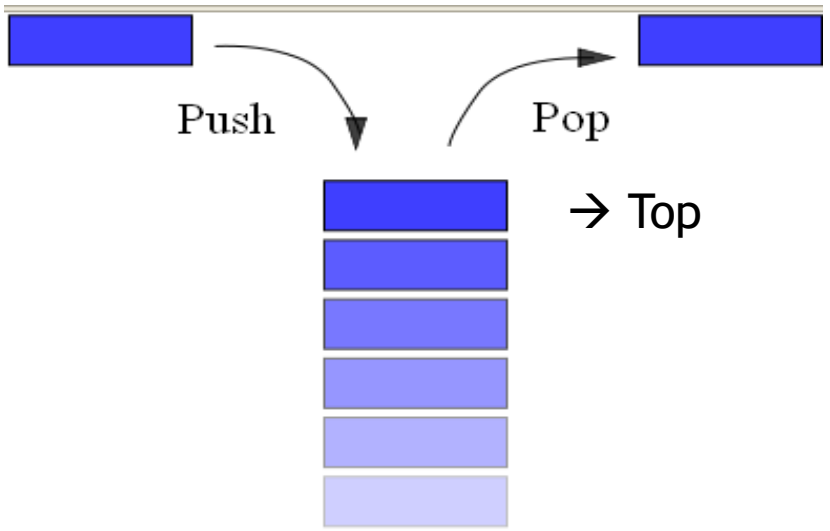
STRUKTUR DATA
KOM 207

Departemen Ilmu Komputer
Fakultas Matematika dan Ilmu Pengetahuan Alam
Institut Pertanian Bogor
2021

Stack



Last In First Out



Contoh: Tumpukan batu bata, tempat penyimpanan kepingan CD

Stack

- Dua Operasi penting dalam algoritme stack (LIFO):
  - PUSH → Menambahkan elemen ke dalam stack (tumpukan)
  - POP → Memindahkan elemen teratas dari stack (tumpukan
- Stack dapat diimplementasikan dengan menggunakan array dan linked list

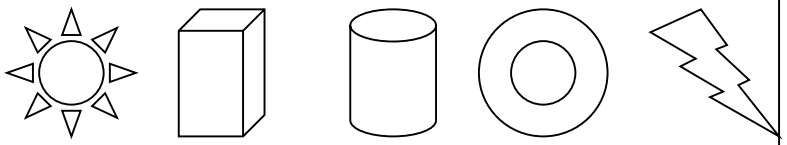
```
struct stack
{
        int item[STACKSIZE];
        int top;
};
```

Implementasi Stack dengan Array → Push

a. Stack Kosong

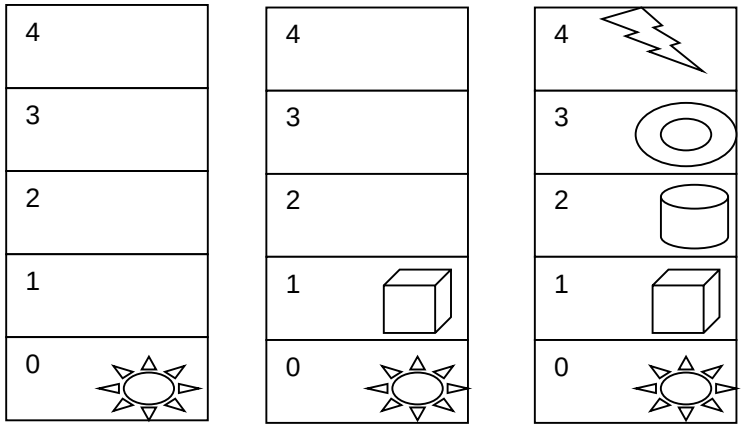
Jumlah elemen yang dapat ditampung oleh stack disamping (MaxElemen)/STACK SIZE = 5
Top of Stack= -1

b. Benda yang akan dimasukkan ke stack



c. Benda masuk ke dalam stack

Top of Stack = 0 | Top of Stack = 1 | Top of Stack = 4

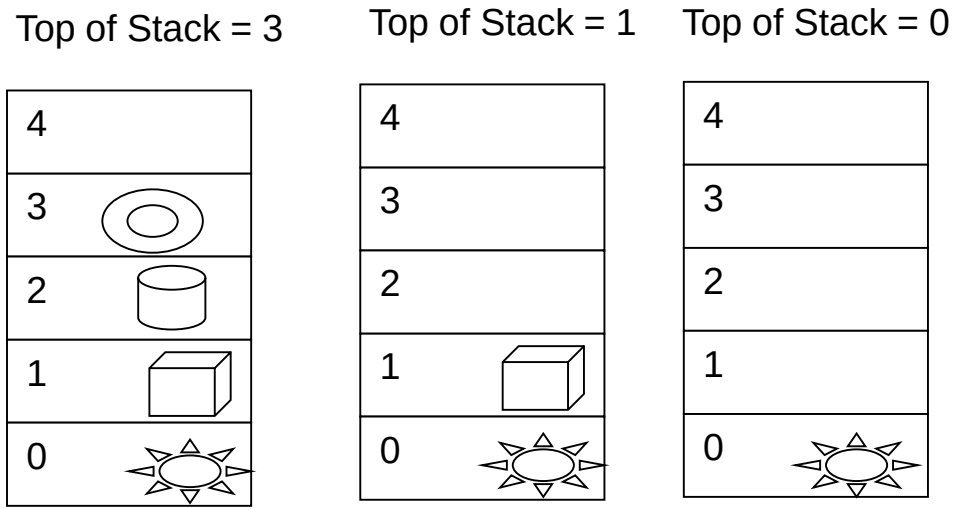



```
void push(struct stack *s,int x)
{
        if(s->top==STACKSIZE-1)
                {
                printf("stack overflow");
                exit(1);
                }
        s->top++;
        s->item[s->top]=x;
}
```

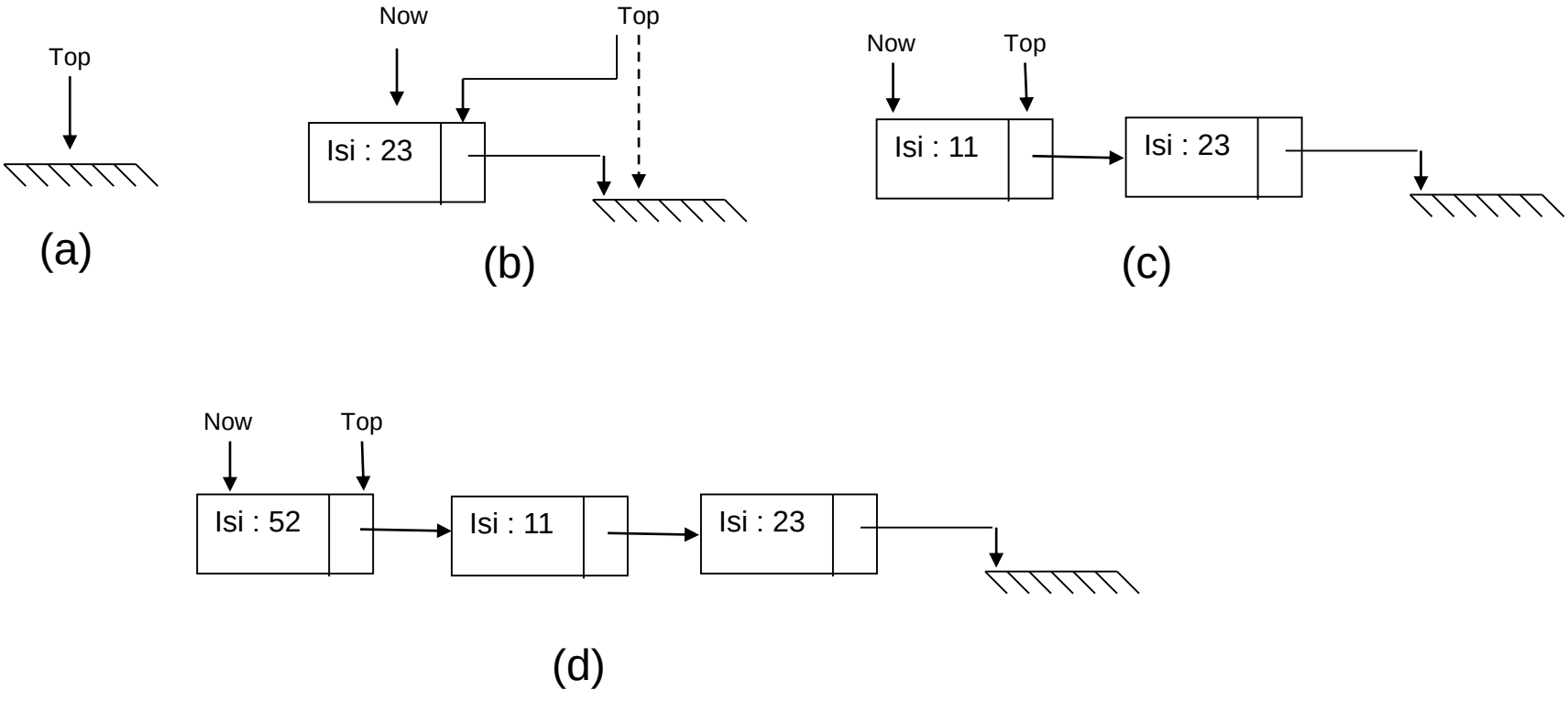
Implementasi Stack dengan Array → Pop

```
int empty(struct stack *s)
{
      if(s->top==-1)
            return TRUE;
      else
            return FALSE;
}

int pop(struct stack *s)
{
      int y;
      if(empty(s))
            {
             printf("Stack Underflows");
             exit(1);
            }
      y=s->item[s->top];
      s->top--;
      return(y);
}
```

d. Benda keluar dari stack

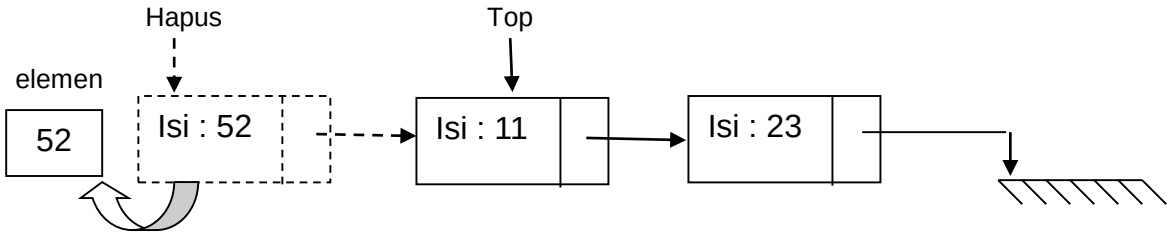


PUSH dengan Linked List

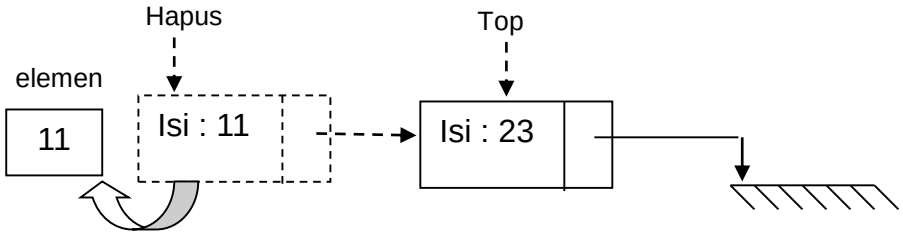


Insert di awal Linked List

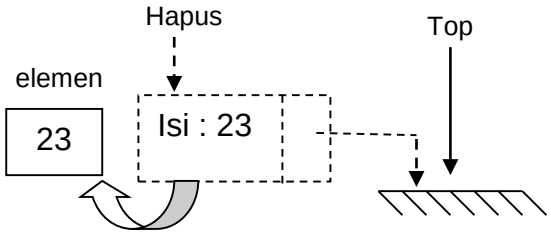
POP dengan Linked List



(a)



(b)

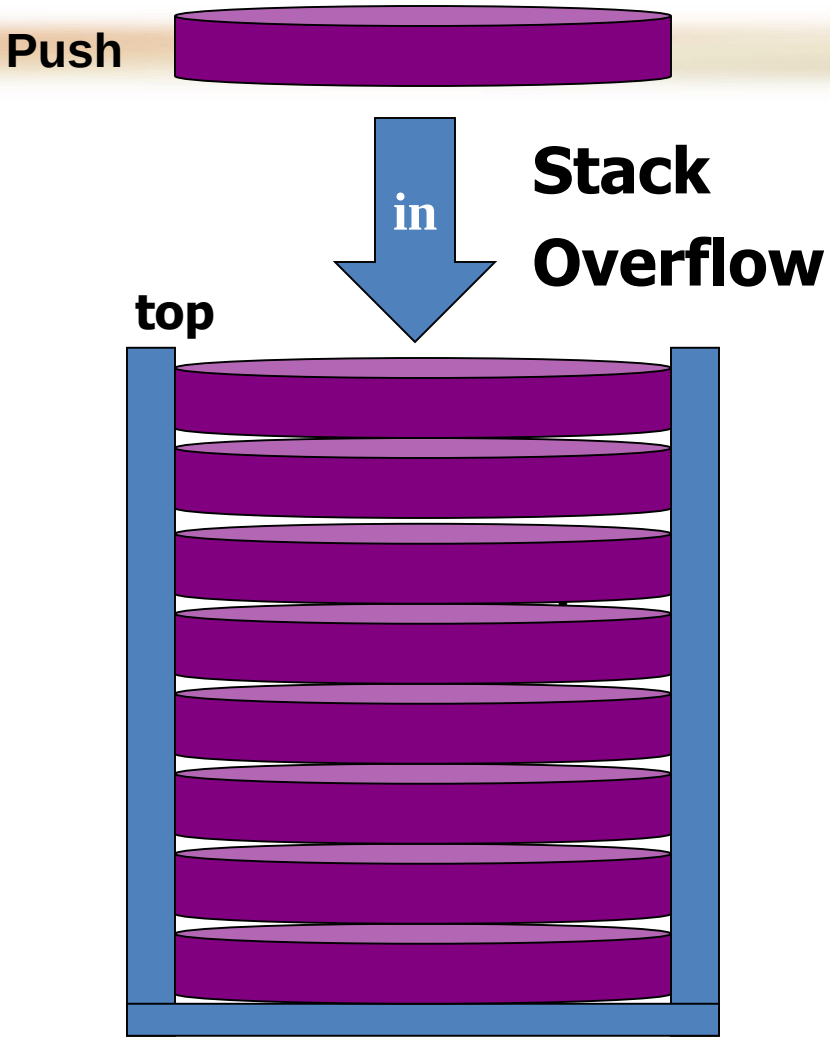


(c)

Delete di awal Linked List

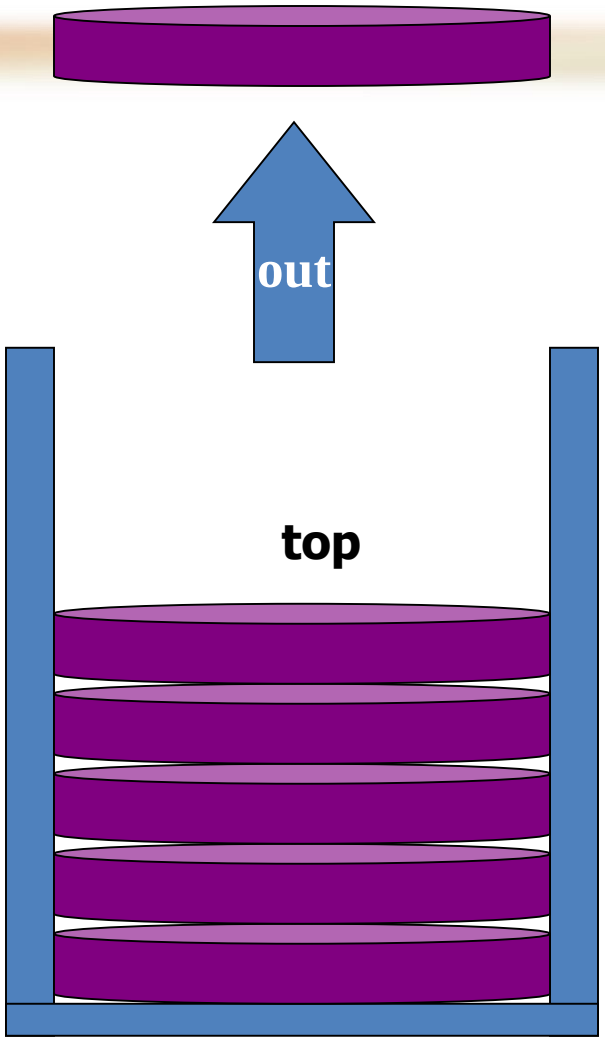
Stack overflow

- Stack overflow Penambahan elemen/data pada sebuah stack yang sudah penuh atau posisi TOP sudah sama dengan kapasitas penyimpanan data
- Penambahan tidak dapat dilakukan Jika Kondisi elemen teratas (Top) sudah sama dengan kapasitas stack.
- Top == Maxsize



Stack underflow

- Stack Underflow
  - Mengambil elemen/data pada sebuah stack yang sudah kosong atau posisi Top sudah sama posisi Bottom
- pengurangan Tidak bisa dilakukan apabila elemen teratas sudah berada pada posisi Bottom
- Top == Bottom



Stack Underflow

QUEUE

- Struktur data antrian
- Dua Operasi penting dalam algoritme QUEUE
  - Enqueue → Penambahan data dilakukan pada ujung sebuah queue
  - Dequeue → penghapusan data dilakukan pada ujung yang lain

  Data yang dihapus adalah data yang paling awal ditambahkan (FIFO)

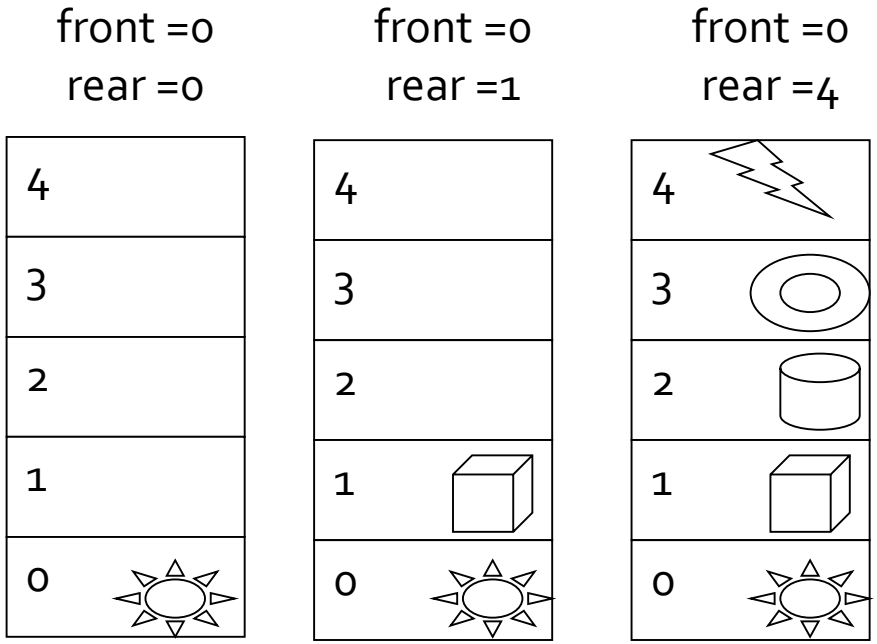
```
struct queue
{
        int item[MAXQUEUE];
        int front;
        int rear;
};
```

```
void init(struct queue *pq)
{
        pq->rear = -1;
        pq->front=0;
}
```

Enqueue

```
void insert(struct queue *pq,int x)
{
      if(pq->rear==MAXQUEUE-1)
            {
             printf("Queue Overflows");
             exit(1);
            }
      pq->rear++;
      pq->item[pq->rear]=x;
}
```

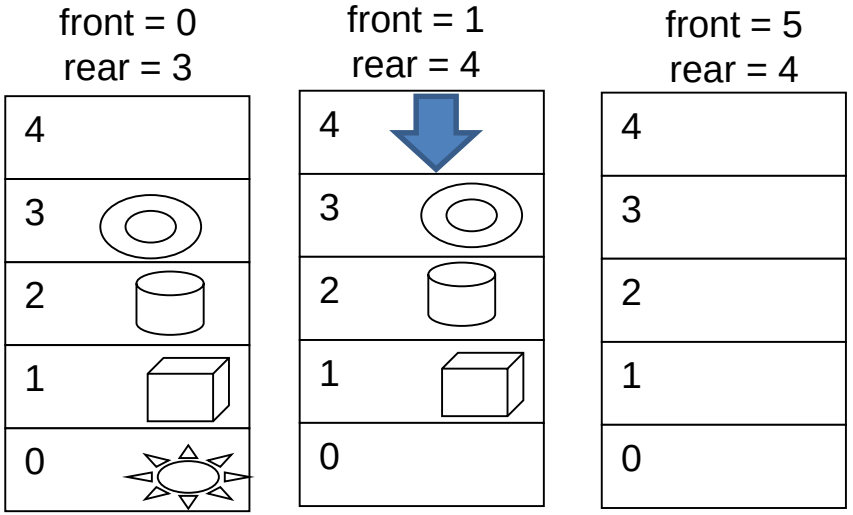
Benda masuk ke dalam Queue



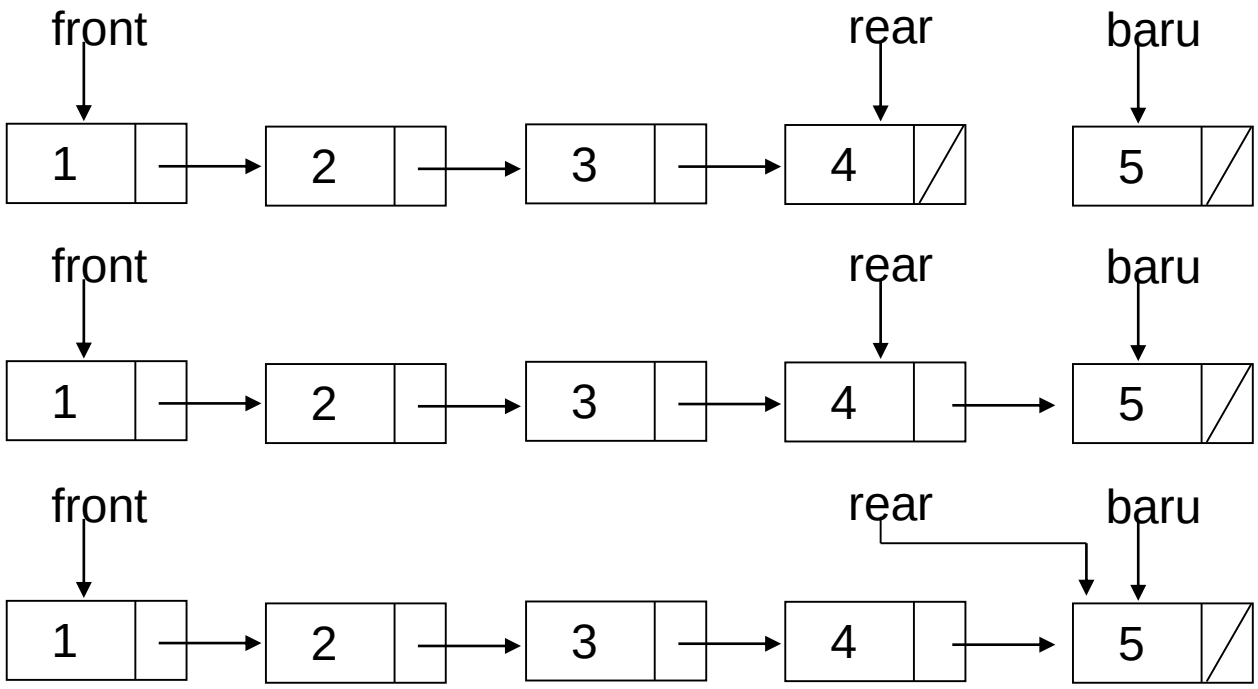
Dequeue

```
int empty(struct queue *pq)
{
        if(pq->front==MAXQUEUE)
                return TRUE;
        else
                return FALSE;
}
int remov(struct queue *pq)
{
        int x;
        if(empty(pq))
        {
                printf("Queue Underflows");
                 exit(1);
        }
        x=pq->item[pq->front];
        pq->front++;
        return x;
}
```

Benda keluar dari queue



Enqueue dengan Linked List



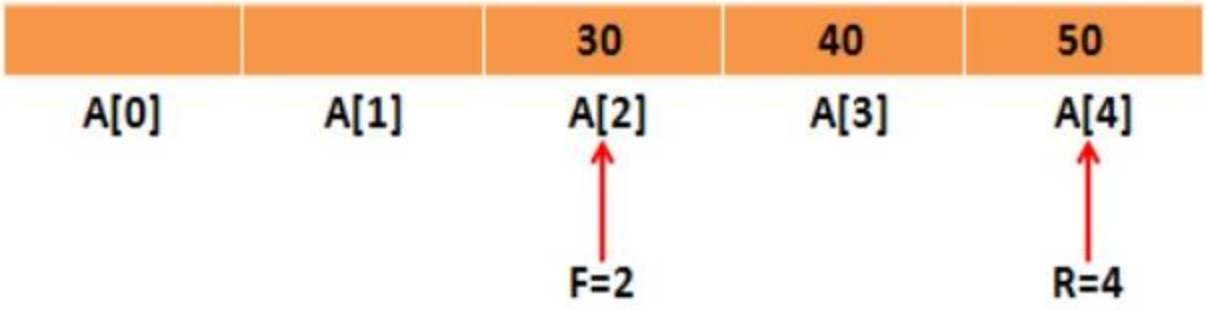
Insert di akhir Linked List

Dequeue dengan Linked List

- Untuk Dequeue dengan Linked List menggunakan hapus awal atau akhir ?

Circular Queue

- Wastage of memory in standard queue in DEQUEUE operations

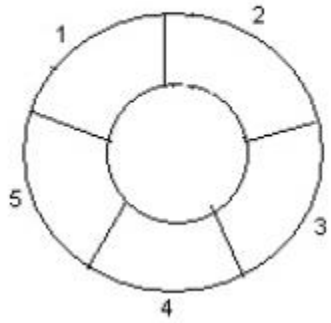


- The last node is connected with first node to make a circle

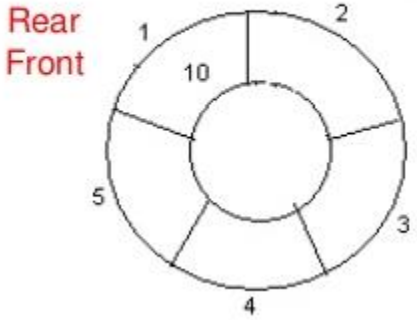
https://www.slideshare.net/JankiShah24/queue-in-data-structure-81323915

Circular Queue

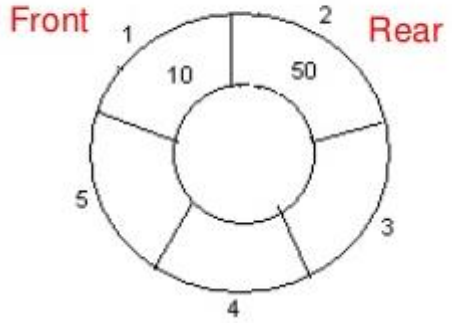
1. Initially, Rear = 0, Front = 0.



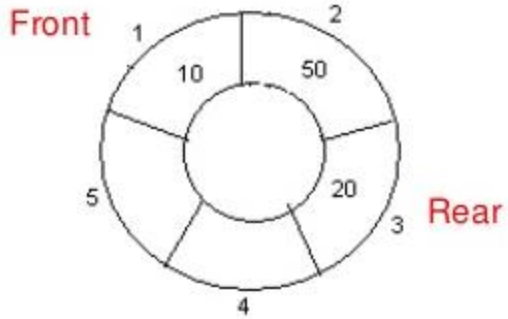
2. Insert 10, Rear = 1, Front = 1.



3. Insert 50, Rear = 2, Front = 1.

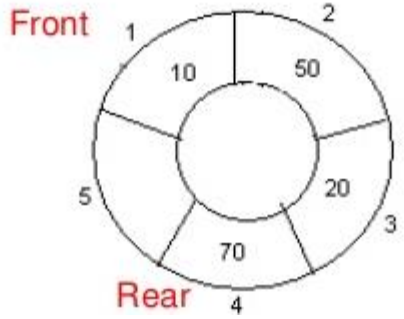


4. Insert 20, Rear = 3, Front = 1.

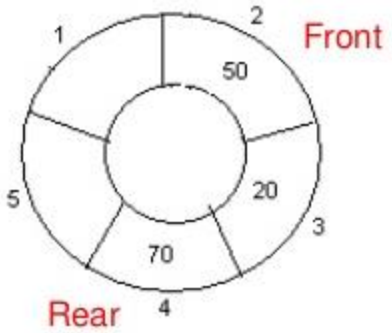


Circular Queue

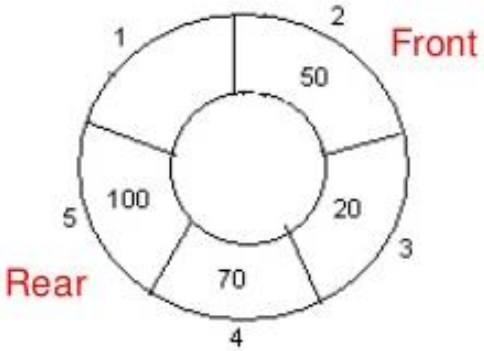
5. Insert 70, Rear = 4, Front = 1.



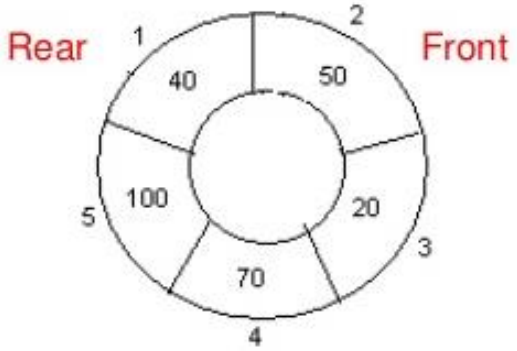
6. Delete front, Rear = 4, Front = 2.



7. Insert 100, Rear = 5, Front = 2.

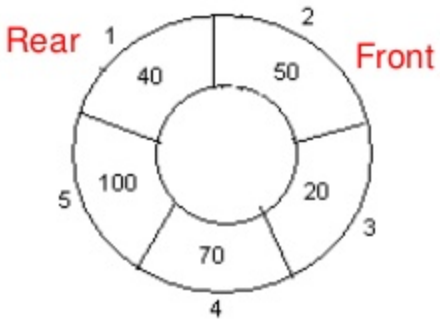


8. Insert 40, Rear = 1, Front = 2.

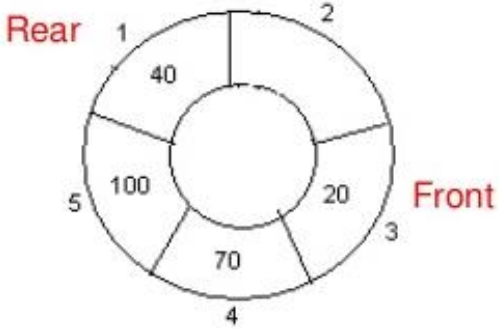


Circular Queue

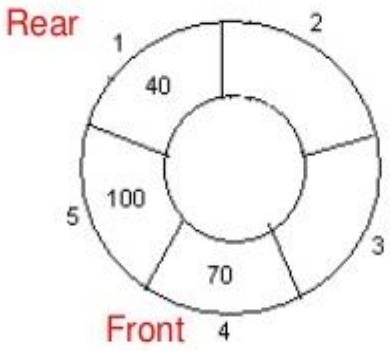
9. Insert 140, Rear = 1, Front = 2.
As Front = Rear + 1, so Queue overflow.



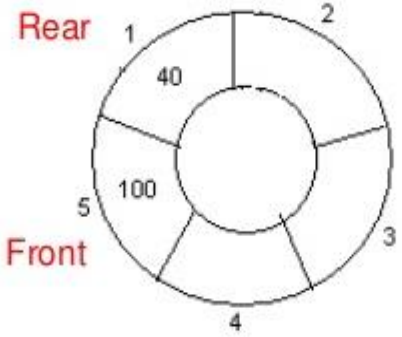
10. Delete front, Rear = 1, Front = 3.



11. Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.



Priority Queue

- A Priority Queue – a different kind of queue.
- Similar to a regular queue:
  - insert in rear,
  - remove from front.
- Items in priority queue are **ordered** by **some key**
- Item with the lowest key / highest key is always at the front from where they are removed.
- Items **then** 'inserted' in 'proper' position
- Idea behind the Priority Queue is simple:
  - Is a queue, **But** the items are **ordered** by a **key**.
  - **Implies your 'position' in the queue may be changed** by the arrival of a new item.

Priority Queue

- Many, many applications.
  - Scheduling queue**s** for a processor, print queues, transmit queues, backlogs, etc…..

- Note: a priority queue is no longer FIFO!
- You will still **remove** from front of queue, but **insertions** are governed by a priority.

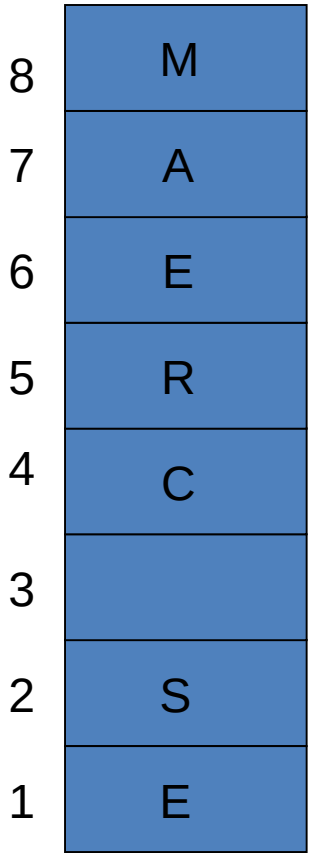
Diskusi

1. Buatlah sebuah program untuk mendeteksi apakah sebuah string merupakan palindrome atau bukan.
2. Buatlah sebuah program untuk mengubah infix menjadi postfix

   https://slideplayer.com/slide/6976809/

Penggunaan Stack 1

Ex: Make a program to check whether a sentence is palindrome or not.



Kata yang dimasukkan : ES CREAM

Proses Pemasukan Kata → Push setiap karakter ke dalam stack

Proses pengeluaran kata → Pop setiap karakter dari stack

Kata hasil pembalikan : MAERC SE

Bukan Palindrome !!!!!

Program Palindrome

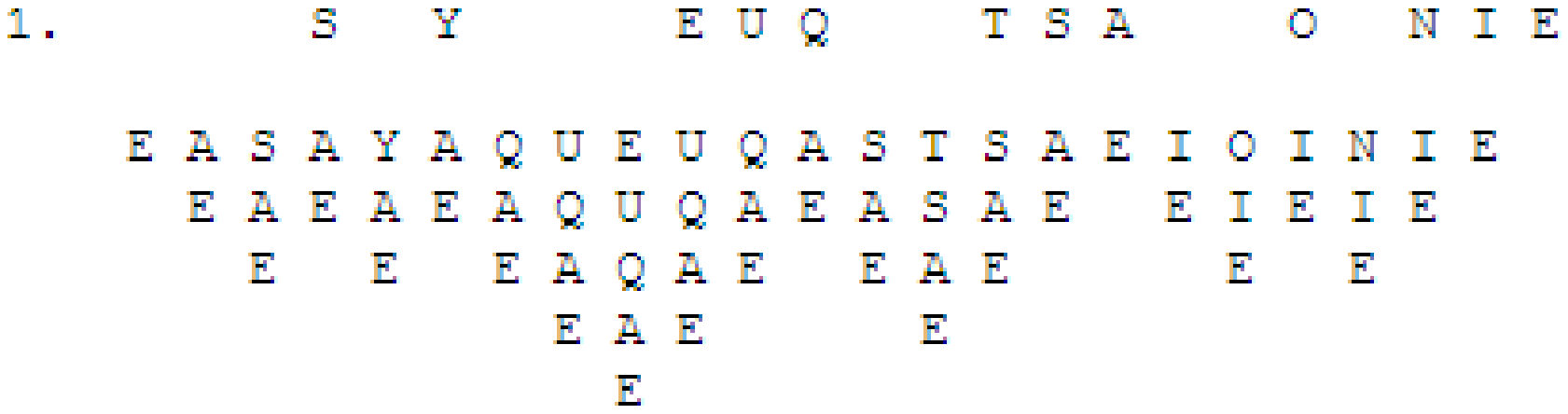
```
begin
  clrscr;
  inisialisasi (T);
  write('Isikan sembarang kalimat :'); readln(Kalimat1);
  panjang := length(Kalimat1);

{***Memasukkan elemen ke dalam stack***}
  for i:= 1 to panjang do
   push (T, Kalimat1[i]);

  writeln('Kalimat setelah dibalik :');
  {***Memasukkan elemen ke dalam stack***}
  for i:= 1 to panjang do
   pop (T, Kalimat2[i]);
  write(Kalimat2);
{*****bandingkan******}
If kalimat1 = kalimat2 then
        write('Palindrome')
Else
        write('Bukan palindrome');
readln;
end.
```

Penggunaan Stack 2

A letter means push and an asterisk means pop in the following sequence. Give the sequence of values returned by the pop operations when this sequence of operations is performed on an initially empty LIFO stack. E A S * Y * Q U E * * * S T * * * I O * N * * *

E A S * Y * Q U E * * * S T * * * I O * N * * *

1.



Penggunaan Queue

- A letter means put and an asterisk means get in the following sequence. Give the sequence of values returned by the get operation when this sequence of operations is performed on an initially empty FIFO queue. E A S * Y * Q U E * * * S T * * * I O * N * * *

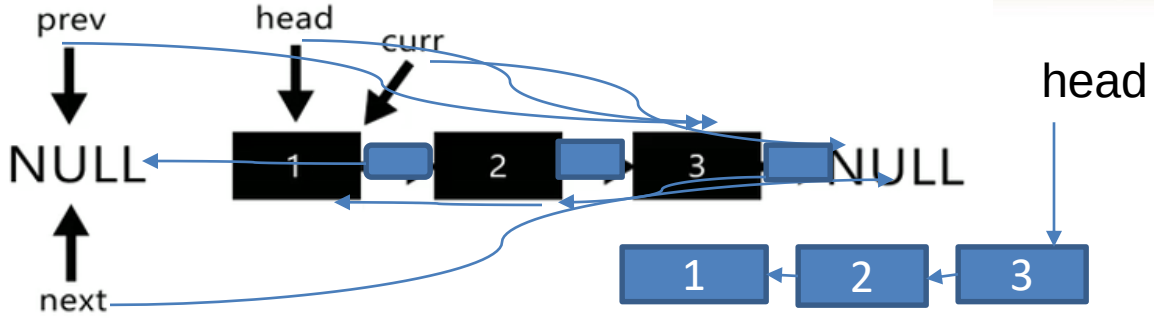
E A S * Y * Q U E * * * S T * * * I O * N * * *

4. E A S Y Q U E S T I O N

E E E A A S S S S S Y Y Y Q U U U E S T T T I O N
A A S S Y Q Q U Q U E E E S T I O N
S Y U E S T T O N
E

Linked List 1

Diketahui kondisi sebuah linked list sebagai berikut:

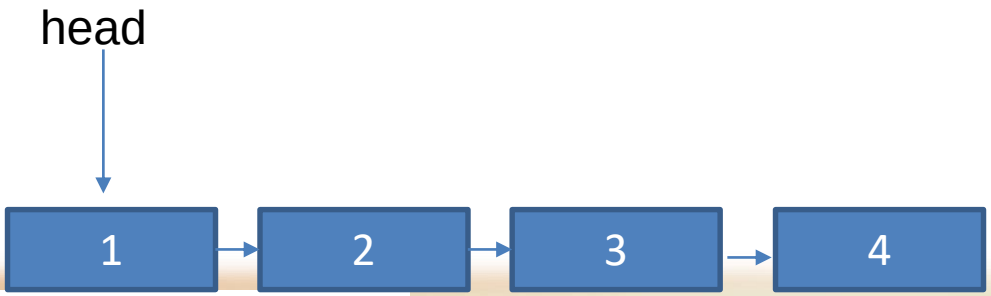


Jika pada linked list tersebut dijalankan potongan program berikut:

```
while (current != NULL)
{
   next = current -> next;
   current -> next = prev;

   prev =current;
   current = next;
}
head = prev;
```

Apakah yang terjadi pada linked list setelah potongan program tersebut selesai dijalankan?

Linked List 2



What does the following function do for a given Linked List with first node as *head*?

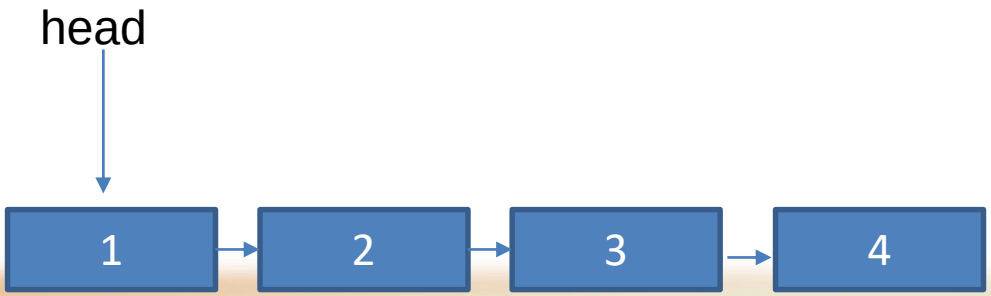
```
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}
```

A. Prints all nodes of linked lists

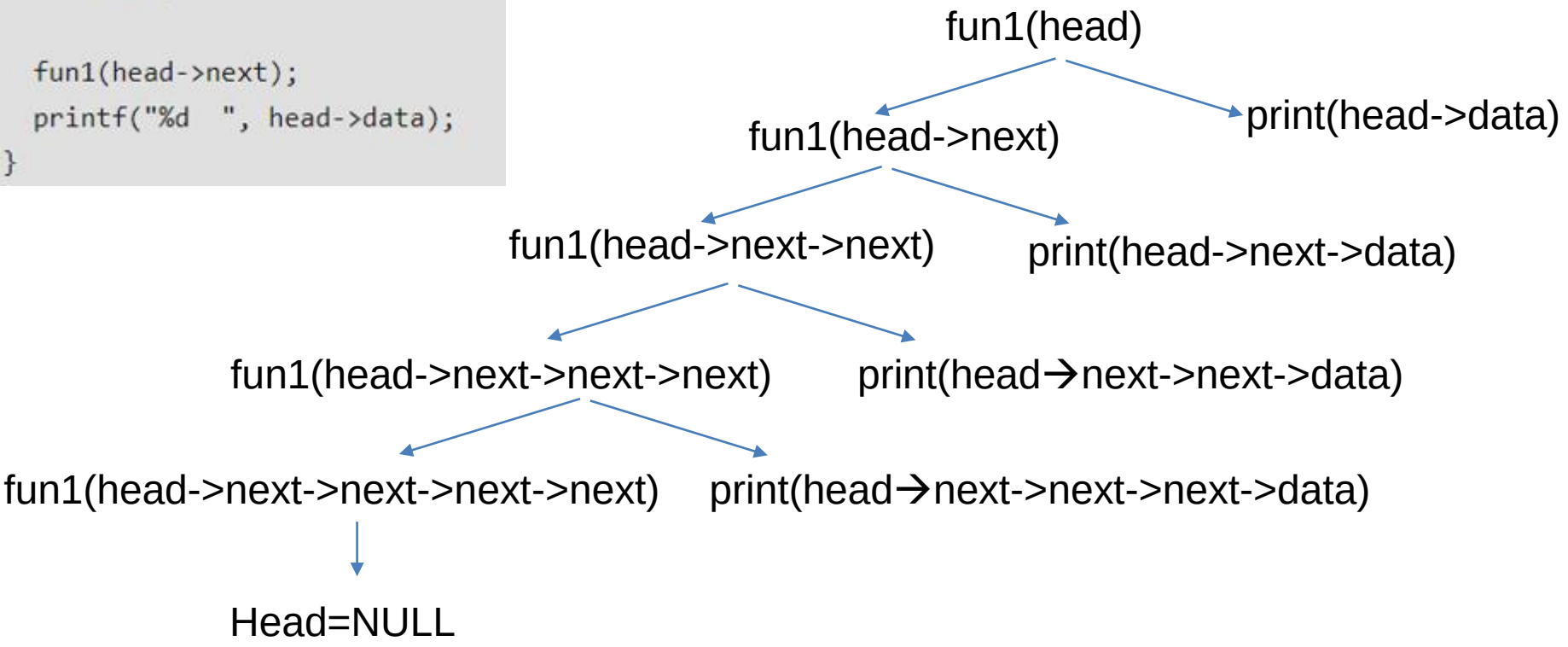
B. Prints all nodes of linked list in reverse order

C. Prints alternate nodes of Linked List

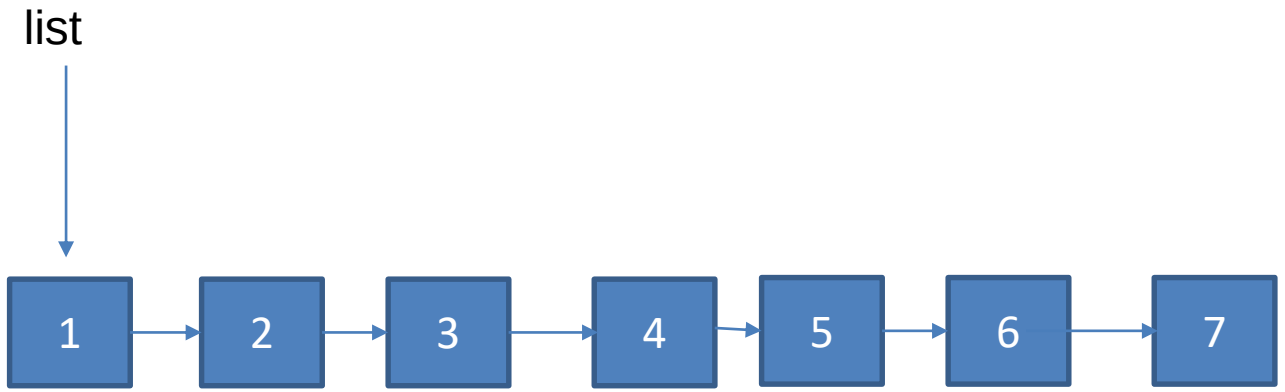
D. Prints alternate nodes in reverse order




```
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}
```



```
struct node
{
  int value;
  struct node *next;
};
void rearrange(struct node *list)
{
  struct node *p, * q;
  int temp;
  if ((!list) || !list->next)
     return;
  p = list;
  q = list->next;
  while(q)
  {
     temp = p->value;
     p->value = q->value;
     q->value = temp;
     p = q->next;
     q = p->next;
  }
}
```



Recursion

Chapter Objectives

- To understand how to think recursively
- To learn how to trace a recursive method
- To learn how to write recursive algorithms and methods for searching arrays
- To learn about recursive data structures and recursive methods for a LinkedList class

Recursive Thinking

- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that would be difficult to solve in other ways
- Recursion splits a problem into one or more simpler versions of itself

FIGURE 7.1
A Set of Nested Wooden Figures

Steps to Design a Recursive Algorithm

- There must be at least one case (**the base case**), for a small value of n, that can be solved directly
- A problem of a given size n can be split into one or more smaller versions of the same problem (**recursive case**)
- Recognize the base case and provide a solution to it
- Devise a strategy to **split the problem into smaller versions of itself** while **making progress toward the base case**
- **Combine the solutions of the smaller problems** in such a way as to **solve the larger problem**

String Length Algorithm

Recursive Algorithm for Finding the Length of a String

1. `if` the string is empty (has no characters)
2. The length is 0.

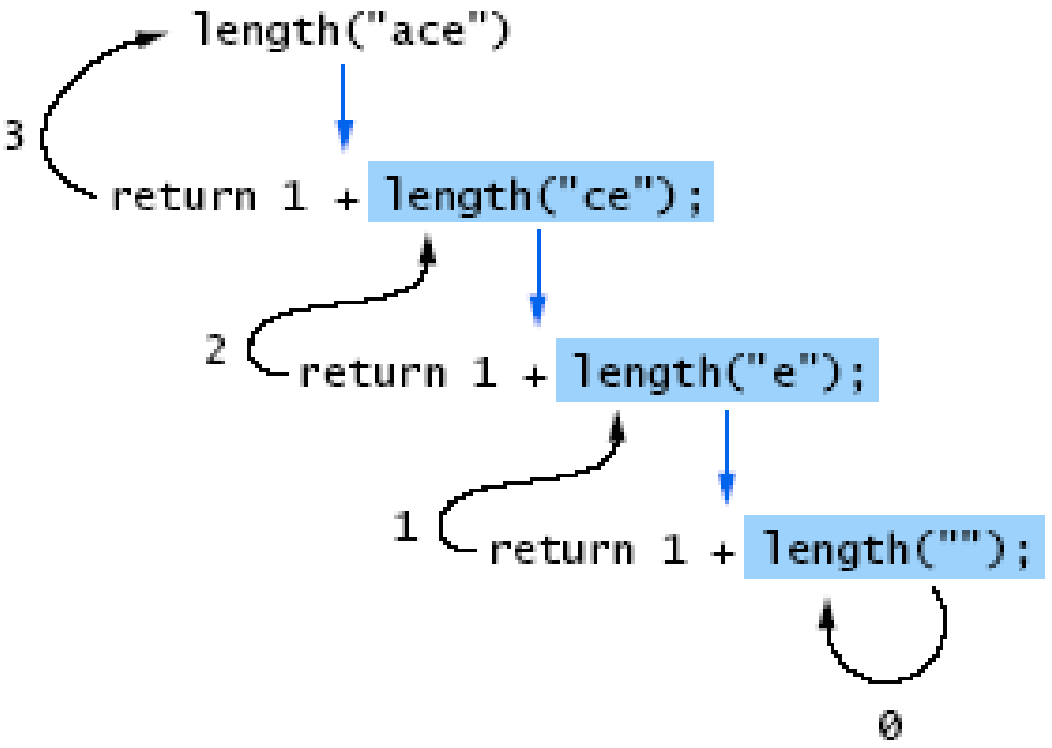
   `else`
3. The length is 1 plus the length of the string that excludes the first character.

Proving that a Recursive Method is Correct

- Proof by induction
  - Prove the theorem is true for the base case
  - Show that if the theorem is assumed true for n, then it must be true for n+1
- Recursive proof is similar to induction
  - Verify the base case is recognized and solved correctly
  - Verify that each recursive case makes progress towards the base case
  - Verify that if all smaller problems are solved correctly, then the original problem is also solved correctly

Tracing a Recursive Method

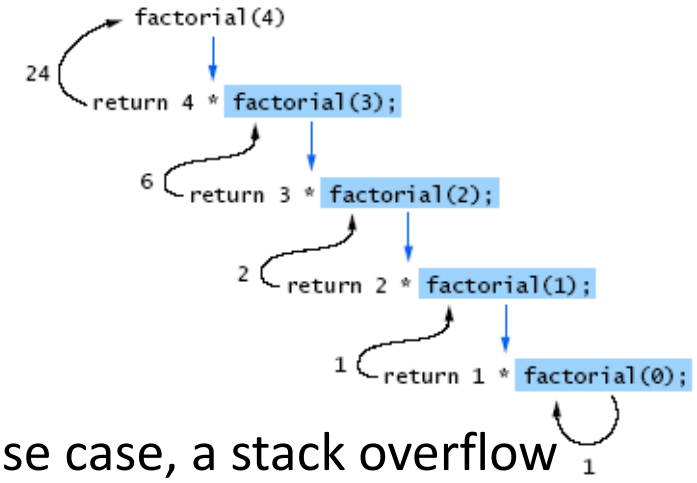
FIGURE 7.2
Trace of
`length("ace")`



Recursive Definitions of Mathematical Formulas

- Mathematicians often use recursive definitions of formulas that lead very naturally to recursive algorithms
- Examples include:
  - Factorial
  - Powers
  - Greatest common divisor

FIGURE 7.5
Trace of `factorial(4)`



- If a recursive function never reaches its base case, a stack overflow error occurs

Recursive Factorial Method

```
// C program to find factorial of given number
#include <stdio.h>

// function to find factorial of given number
unsigned int factorial(unsigned int n)
{
        if (n == 0)
           return 1;
        return n * factorial(n - 1);
}

int main()
{
        int num = 5;
        printf("Factorial of %d is %d", num, factorial(num));
        return 0;
}
```

Source: https://www.geeksforgeeks.org/program-for-factorial-of-a-number/

Recursive Algorithm for Calculating x^n

```
/* Function to calculate x raised to the power y */
int power(int x, unsigned int y)
{
        if (y == 0)
                return 1;
        else if (y%2 == 0)
                return power(x, y/2)*power(x, y/2);
        else
                return x*power(x, y/2)*power(x, y/2);
}

/* Program to test function power */
int main()
{
        int x = 2;
        unsigned int y = 3;

        printf("%d", power(x, y));
        return 0;
}
```

Source: https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/

Recursion Versus Iteration

Recursion

- Terminates when a base case is reached.
- Each recursive call requires extra space on the stack frame (memory).
- If we get infinite recursion, the program may run out of memory and result in stack overflow.
- Solutions to some problems are easier to formulate recursively.

Iteration

- Terminates when a condition is proven to be false.
- Each iteration does not require extra space.
- An infinite loop could loop forever since there is no extra memory being created.
- Iterative solutions to a problem may not always be as obvious as a recursive solution.

Iterative Factorial Method

```
// function to find factorial of given number
unsigned int factorial(unsigned int n)
{
        int res = 1, i;
        for (i = 2; i <= n; i++)
                res *= i;
        return res;
}

int main()
{
        int num = 5;
        printf(
                "Factorial of %d is %d", num, factorial(num));
        return 0;
}
```

Source: https://www.geeksforgeeks.org/program-for-factorial-of-a-number/

Efficiency of Recursion

- Recursive methods often have slower execution times when compared to their iterative counterparts
- The overhead for loop repetition is smaller than the overhead for a method call and return
- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
  - The reduction in efficiency does not outweigh the advantage of readable code that is easy to debug

An Exponential Recursive Fibonacci Method

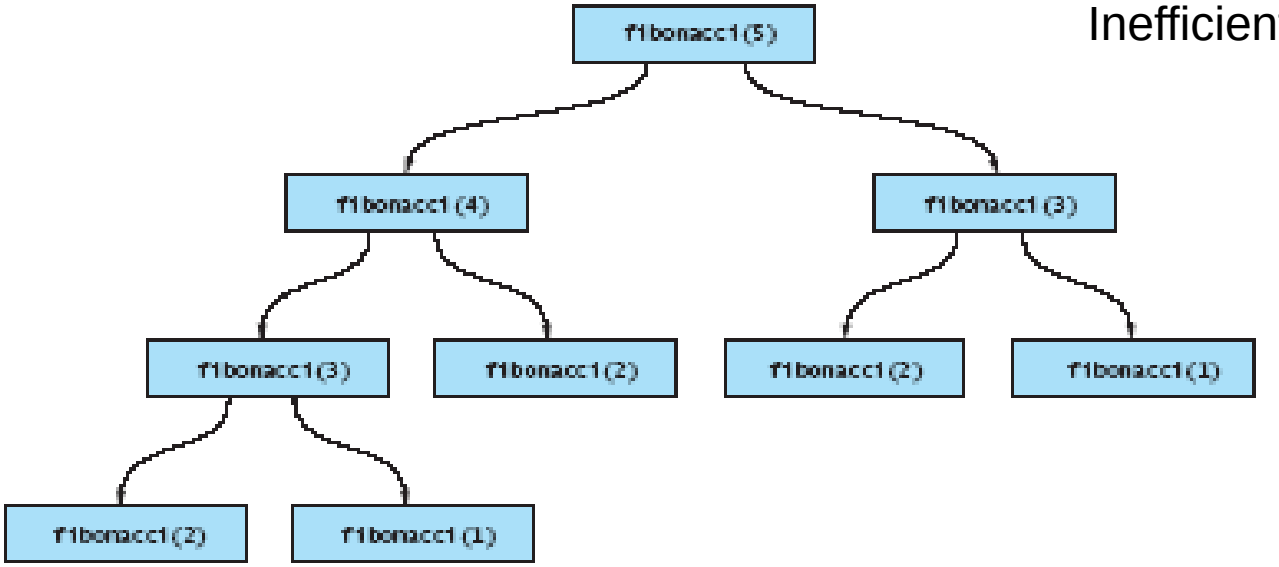
```
/** Recursive method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param n The position of the Fibonacci number being calculated
    @return The Fibonacci number
*/
public static int fibonacci(int n) {
    if (n <= 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

An O(n) Recursive Fibonacci Method

```
/** Recursive O(n) method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param fibCurrent The current Fibonacci number
    @param fibPrevious The previous Fibonacci number
    @param n The count of Fibonacci numbers left to calculate
    @return The value of the Fibonacci number calculated so far
*/
private static int fibo(int fibCurrent, int fibPrevious, int n) {
    if (n == 1)
        return fibCurrent;
    else
        return fibo(fibCurrent + fibPrevious, fibCurrent, n - 1);
}
```

Efficiency of Recursion: Exponential Fibonacci

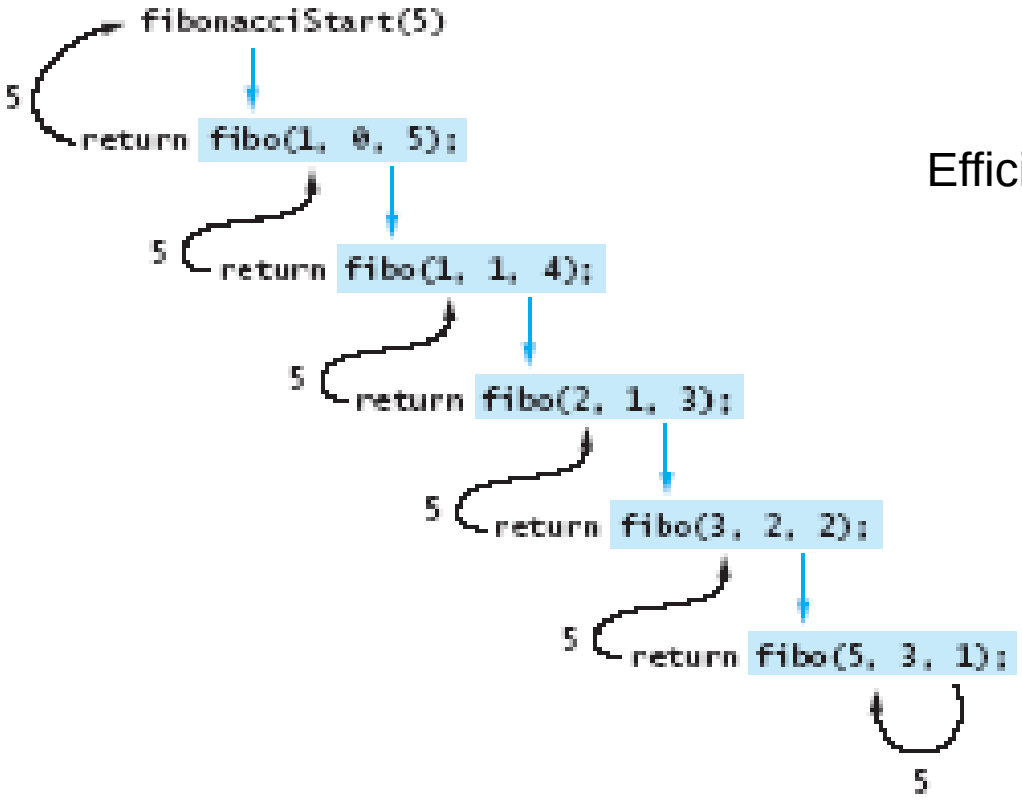
FIGURE 7.6
Method Calls Resulting from `fibonacci(5)`



Inefficient

Efficiency of Recursion: O(n) Fibonacci

FIGURE 7.7
Trace of `fibonacciStart(5)`



Efficient

Recursive Array Search

- Searching an array can be accomplished using recursion
- Simplest way to search is a linear search
  - Examine one element at a time starting with the first element and ending with the last
- Base case for recursive search is an empty array
  - Result is negative one
- Another base case would be when the array element being examined matches the target
- Recursive step is to search the rest of the array, excluding the element just examined

Algorithm for Recursive Linear Array Search

Algorithm for Recursive Linear Array Search

1. `if` the array is empty
2. The result is –1.

   `else if` the first element matches the target
3. The result is the subscript of the first element.

   `else`
4. Search the array excluding the first element and return the result.

Implementation of Recursive Linear Search

```c
#define SZ 15
int linearSearch(int* list, int key, int idx){
    if(idx == -1)
        return -1;
    else if(list[idx] == key)
        return idx;
    else{
        return linearSearch(list, key, idx-1);
    }
}

int main(){
    int list[SZ] = {2, 5, 12, 6, 10, 0, 3, 9, 15, 23, 11, 7, 3, 12, 23};
    int idx;
    idx = linearSearch(list, 0, SZ-1);
    if(idx == -1)
        printf("Data is not found!\n");
    else
        printf("Data is available at %d\n", idx);
}
```

Iterative Sum of Array

```
int IterativeArraySum(int *A, int n){
    int sum = 0;
    int i = 0;

    for(i=0; i<n; i++){
        sum = sum + A[i];
    }
    return sum;
```

Recursive Sum of Array

```
#include <stdio.h>

int arraySum(int* A, int n){
    if(n==0)
        return(A[0]);
    else
        return(A[n] + arraySum(A, n-1));
}
```

Design of a Binary Search Algorithm

- Binary search can be performed only on an array that has been sorted
- Stop cases
  - The array is empty
  - Element being examined matches the target
- Checks the middle element for a match with the target
- Throw away the half of the array that the target cannot lie within

Design of a Binary Search Algorithm (continued)

Binary Search Algorithm

1. `if` the array is empty
2. Return –1 as the search result.

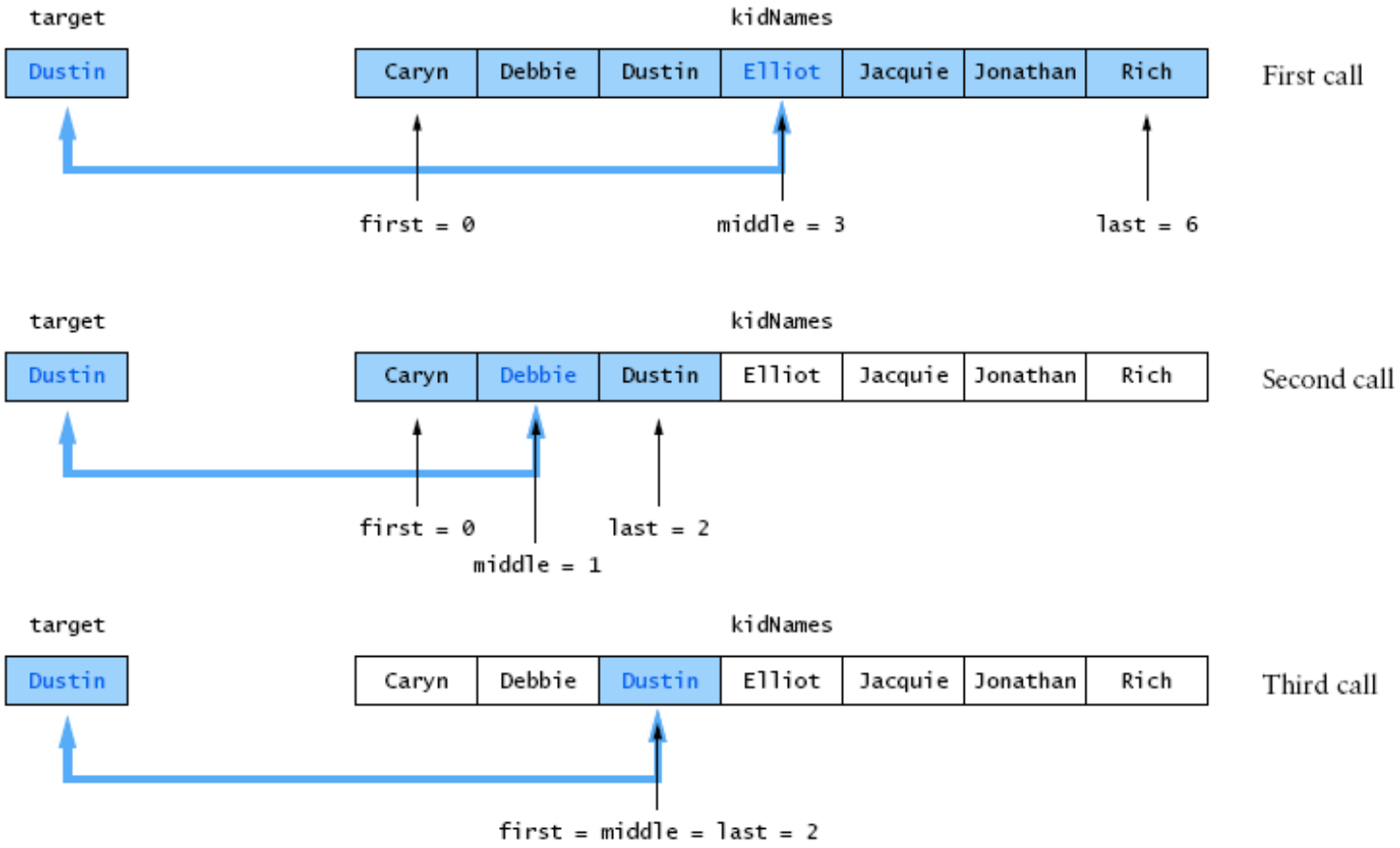
   `else if` the middle element matches the target
3. Return the subscript of the middle element as the result.

   `else if` the target is less than the middle element
4. Recursively search the array elements before the middle element and return the result.

   `else`
5. Recursively search the array elements after the middle element and return the result.

Design of a Binary Search Algorithm (continued)

FIGURE 7.9
Binary Search for "Dustin"



Implementation of a Binary Search Algorithm

```
// C program to implement recursive Binary Search
// A recursive binary search function. It returns  location of x in given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
        if (r >= l) {
                int mid = l + (r - l) / 2;

                // If the element is present at the middle itself
                if (arr[mid] == x)
                        return mid;

                // If element is smaller than mid, then  it can only be present in left subarray
                if (arr[mid] > x)
                        return binarySearch(arr, l, mid - 1, x);

                // Else the element can only be present  in right subarray
                return binarySearch(arr, mid + 1, r, x);
        }

        // We reach here when element is not
        // present in array
        return -1;
}
```

Source: https://www.geeksforgeeks.org/binary-search/

Implementation of a Binary Search Algorithm

```
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present in array") : printf("Element is present at index %d", result);
    return 0;
}
```

Source: https://www.geeksforgeeks.org/binary-search/

Efficiency of Binary Search and the Comparable Interface

- At each recursive call we eliminate half the array elements from consideration
  - $O(\log_2 n)$
- Classes that implement the Comparable interface must define a compareTo method that enables its objects to be compared in a standard way
  - CompareTo allows one to define the ordering of elements for their own classes

Recursive Data Structures

- Computer scientists often encounter data structures that are defined recursively
  - Trees are defined recursively
- Linked list can be described as a recursive data structure
- Recursive methods provide a very natural mechanism for processing recursive data structures
- The first language developed for artificial intelligence research was a recursive language called LISP

Recursive Definition of a Linked List

- A non-empty linked list is a collection of nodes such that each node references another linked list consisting of the nodes that follow it in the list
- The last node references an empty list
- A linked list is empty, or it contains a node, called the list head, that stores data and a reference to a linked list

Recursive Size Method

```
/** Finds the size of a list.
    @param head The head of the current list
    @return The size of the current list
*/
private int size(Node<E> head) {
    if (head == null)
        return 0;
    else
        return 1 + size(head.next);
}

/** Wrapper method for finding the size of a list.
    @return The size of the list
*/
public int size() {
    return size(head);
}
```

Recursive Replace Method

```
/** Replaces all occurrences of oldObj with newObj.
    post: Each occurrence of oldObj has been replaced by newObj.
    @param head The head of the current list
    @param oldObj The object being removed
    @param newObj The object being inserted
*/
private void replace(Node<E> head, E oldObj, E newObj) {
    if (head != null) {
        if (oldObj.equals(head.data))
            head.data = newObj;
        replace(head.next, oldObj, newObj);
    }
}

    /*  Wrapper method for replacing oldObj with newObj.
        post: Each occurrence of oldObj has been replaced by newObj.
        @param oldObj The object being removed
        @param newObj The object being inserted
    */
    public void replace(E oldObj, E newObj) {
        replace(head, oldObj, newObj);
    }
```

Recursive Add Method

```
/** Adds a new node to the end of a list.
    @param head The head of the current list
    @param data The data for the new node
*/
private void add(Node<E> head, E data) {
    // If the list has just one element, add to it.
    if (head.next == null)
        head.next = new Node<E>(data);
    else
        add(head.next, data);      // Add to rest of list.
}

/** Wrapper method for adding a new node to the end of a list.
    @param data The data for the new node
*/
public void add(E data) {
    if (head == null)
        head = new Node<E>(data);  // List has 1 node.
    else
        add(head, data);
}
```

Recursive Remove Method

```
/** Removes a node from a list.
    post: The first occurrence of outData is removed.
    @param head The head of the current list
    @param pred The predecessor of the list head
    @param outData The data to be removed
    @return true if the item is removed
            and false otherwise
*/
private boolean remove(Node<E> head, Node<E> pred, E outData) {
    if (head == null)  // Base case - empty list.
        return false;
    else if (head.data.equals(outData)) {  // 2nd base case.
        pred.next = head.next;  // Remove head.
        return true;
    } else
        return remove(head.next, head, outData);
}
```

Recursive Remove Method (continued)

```
/** Wrapper method for removing a node (in LinkedListRec).
    post: The first occurrence of outData is removed.
    @param outData The data to be removed
    @return true if the item is removed,
            and false otherwise
*/
public boolean remove(E outData) {
    if (head == null)
        return false;
    else if (head.data.equals(outData)) {
        head = head.next;
        return true;
    } else
        return remove(head.next, head, outData);
}
```

Towers of Hanoi

FIGURE 7.11

Children's Version of Towers of Hanoi

Towers of Hanoi (continued)

TABLE 7.1

Inputs and Outputs for Towers of Hanoi Problem

Problem Inputs
Number of disks (an integer)
Letter of starting peg: L (left), M (middle), or R (right)
Letter of destination peg (L, M, or R), but different from starting peg
Letter of temporary peg (L, M, or R), but different from starting peg and destination peg
Problem Outputs
A list of moves

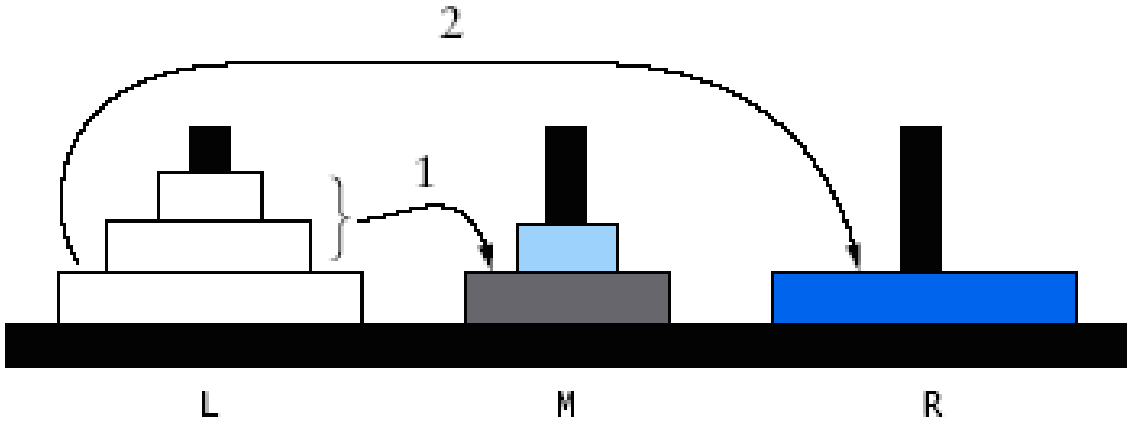
Algorithm for Towers of Hanoi

Solution to 3-Disk Problem: Move 3 Disks from Peg L to Peg R

1. Move the top two disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top two disks from peg M to peg R.

FIGURE 7.12

Towers of Hanoi After the First Two Steps in Solution of the Three-Disk Problem



Algorithm for Towers of Hanoi (continued)

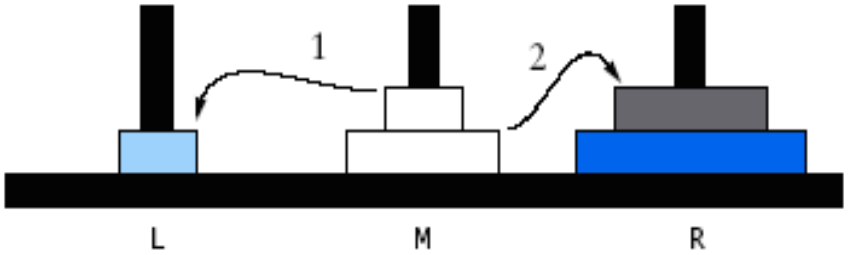
Solution to 2-Disk Problem: Move Top 2 Disks from Peg M to Peg R

1. Move the top disk from peg M to peg L.
2. Move the bottom disk from peg M to peg R.
3. Move the top disk from peg L to peg R.

In Figure 7.13 we show the pegs after steps 1 and 2. When step 3 is completed, the three pegs will be on peg R.

FIGURE 7.13

Towers of Hanoi After First Two Steps in Solution of Two-Disk Problem



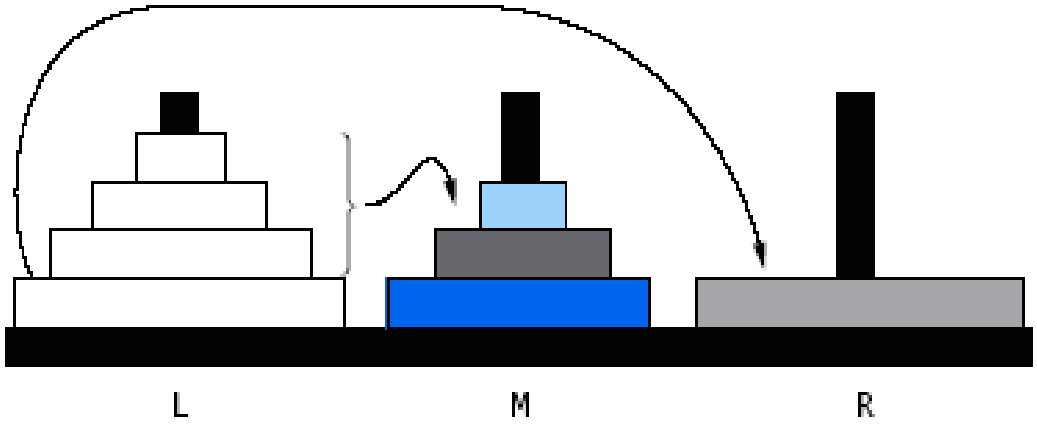
Algorithm for Towers of Hanoi (continued)

Solution to 4-Disk Problem: Move 4 Disks from Peg L to Peg R

1. Move the top three disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top three disks from peg M to peg R.

FIGURE 7.14

Towers of Hanoi After the First Two Steps in Solution of the Four-Disk Problem



Recursive Algorithm for Towers of Hanoi

Recursive Algorithm for *n*-Disk Problem: Move *n* Disks from the Starting Peg to the Destination Peg

1. `if` *n* is 1
2. Move disk 1 (the smallest disk) from the starting peg to the destination peg.
3. `else`
4. Move the top $n - 1$ disks from the starting peg to the temporary peg (neither starting nor destination peg).
5. Move disk *n* (the disk at the bottom) from the starting peg to the destination peg.
6. Move the top $n - 1$ disks from the temporary peg to the destination peg.

Implementation of Recursive Towers of Hanoi

```
/** Class that solves Towers of Hanoi problem. */
public class TowersOfHanoi {
    /** Recursive method for "moving" disks.
        pre: startPeg, destPeg, tempPeg are different.
        @param n is the number of disks
        @param startPeg is the starting peg
        @param destPeg is the destination peg
        @param tempPeg is the temporary peg
        @return A string with all the required disk moves
    */
    public static String showMoves(int n, char startPeg,
                                   char destPeg, char tempPeg) {
        if (n == 1) {
            return "Move disk 1 from peg " + startPeg +
                   " to peg " + destPeg + "\n";
        } else {  // Recursive step
            return showMoves(n - 1, startPeg, tempPeg, destPeg)
                   + "Move disk " + n + " from peg " + startPeg
                   + " to peg " + destPeg + "\n"
                   + showMoves(n - 1, tempPeg, destPeg, startPeg);
        }
    }
}
```

Chapter Review

- A recursive method has a standard form
- To prove that a recursive algorithm is correct, you must
  - Verify that the base case is recognized and solved correctly
  - Verify that each recursive case makes progress toward the base case
  - Verify that if all smaller problems are solved correctly, then the original problem must also be solved correctly
- The run-time stack uses activation frames to keep track of argument values and return points during recursive method calls

Chapter Review (continued)

- Mathematical Sequences and formulas that are defined recursively can be implemented naturally as recursive methods
- Recursive data structures are data structures that have a component that is the same data structure
- Towers of Hanoi and counting cells in a blob can both be solved with recursion
- Backtracking is a technique that enables you to write programs that can be used to explore different alternative paths in a search for a solution

Sorting

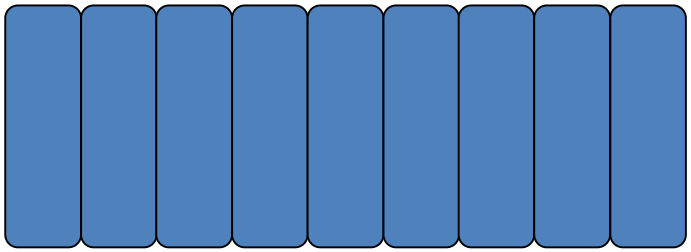
Outline

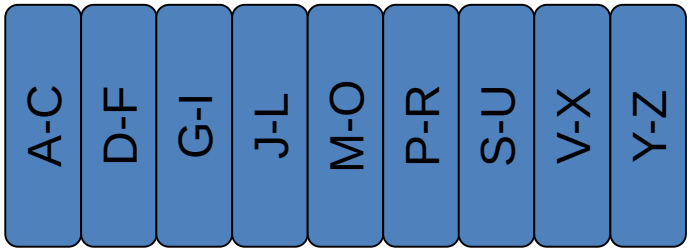
- Pembagian algoritma sorting
- Algoritma sorting
  - Paradigma
  - Contoh
  - Running Time

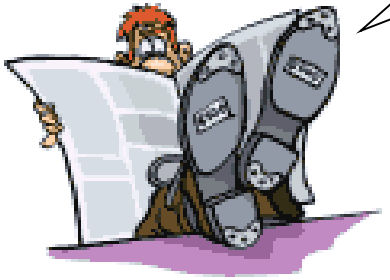
Sorting

- Sorting = pengurutan
- Sorted = terurut menurut kaidah tertentu
- Data pada umumnya disajikan dalam bentuk sorted
- Why?

- Faster and easier in accessing data →find “L”!







Efficient job !

Why Sorting?

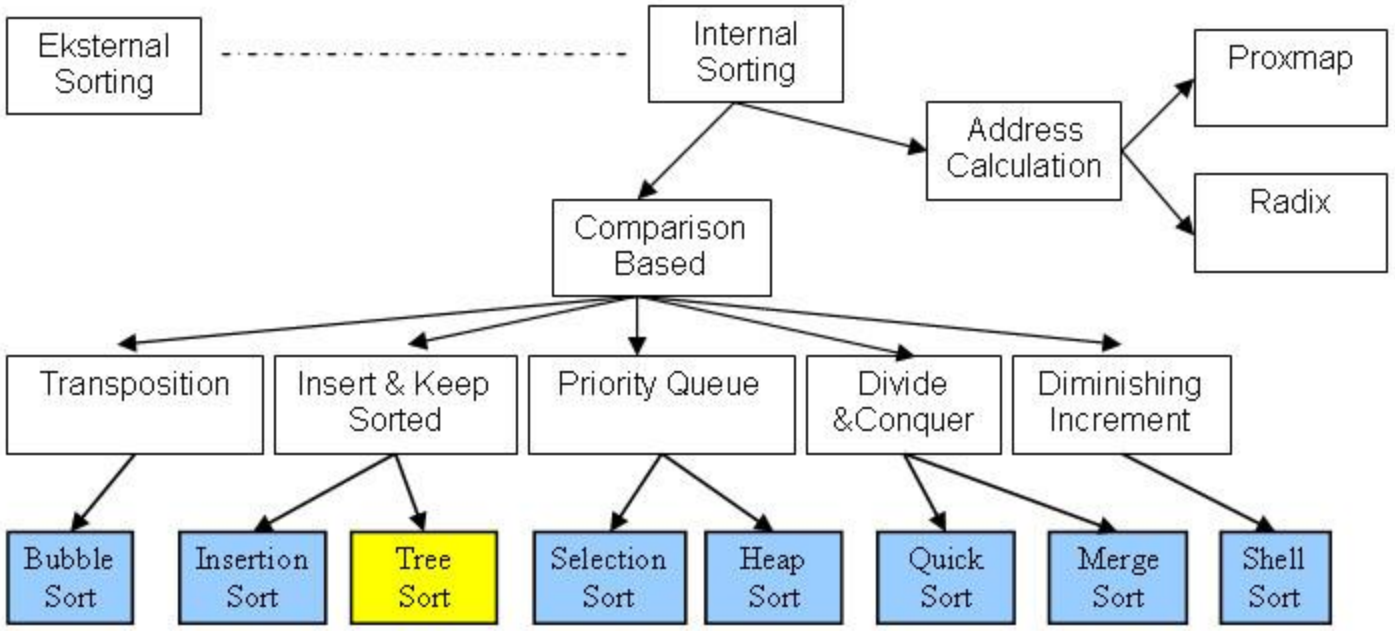
- Menyusun sekelompok elemen data yang tidak terurut menjadi terurut berdasarkan suatu kriteria tertentu.
- Mempermudah dan mempercepat proses pencarian data
- Jika pencarian data mudah, maka proses manipulasi data juga akan lebih cepat.

Internal Vs External

- Internal sorting: refers to the sorting of an array of data that is in RAM
- External sorting: refer to sorting methods that are employed when the data to be sorted is too large to fit in primary memory.
- a sorting algorithm sorts in place if only a constant number of elements of the input array are ever stored outside the array

Metode Sorting

- Metode Sorting berdasarkan kriteria sorting yang digunakan dibedakan menjadi :



Transposition

- Didasarkan pada perbandingan elemen dan pertukaran posisi elemen
- Bubble Sort

Insert & Keep Sorted

- Pemasukan sekumpulan data yang belum terurut ke dalam sekumpulan data yang sudah terurut.
- Mempertahankan keterurutan data yang sudah ada sebelumnya
- Insertion Sort, Tree Sort

Priority Queue

- Cari elemen yang sesuai dengan kriteria pencarian dari seluruh elemen yang ada (elemen prioritas).
- Tempatkan pada posisi yang sesuai
- Ulangi sampai semua elemen telah terurut
- Selection Sort, Heap Sort

Divide & Conquer

- Pecah masalah ke dalam sub-sub masalah
- Sort masing-masing sub masalah
- Gabungkan masing-masing bagian
- Merge Sort, Quick Sort

Diminishing Increment

- Penukaran tempat sepasang elemen dengan jarak tertentu.
- Jarak antar elemen akan terus berkurang sampai dihasilkan keadaan terurut.
- Shell Sort

Addres Calculation

- Membuat pemetaan atas key yang ingin disortir, yang akan mengirimkan key tersebut ke lokasi yang paling mendekati final di output array
- Proxmap Sort dan Radix Sort

Algoritme #1: BUBBLE SORT

- Ide: bubble = busa/udara dalam air
- How?
- Busa dalam air akan naik ke atas. Ketika busa naik ke atas, maka air yang di atasnya akan turun memenuhi tempat bekas busa tersebut.
- Keuntungan:
  - Sederhana dan mudah diimplementasikan
  - Cepat jika data masukan sudah terurut
- Kelemahan:
  - Secara umum membutuhkan waktu proses lebih lama

BUBBLE SORT

- First Pass:

(5 1 4 2 8) (1 5 4 2 8) tukar

(1 5 4 2 8) (1 4 5 2 8) tukar

(1 4 5 2 8) (1 4 2 5 8) tukar

(1 4 2 5 8) (1 4 2 5 8) elemen terakhir jadi yg terbesar

- Second Pass :

(1 4 2 5 8) (1 4 2 5 8)

(1 4 2 5 8) (1 2 4 5 8) tukar

(1 2 4 5 8) (1 2 4 5 8) 2 elemen terakhir sudah terurut

- Third Pass:

(1 2 4 5 8) (1 2 4 5 8)

(1 2 4 5 8) (1 2 4 5 8) 3 elemen terakhir sudah terurut

- Fourth Pass:

(1 2 4 5 8) (1 2 4 5 8) 4 elemen terakhir sudah terurut

- Fifth Pass : Completed

(1 2 4 5 8)

Algoritme #1: BUBBLE SORT

```
void bubbleSort(LIST x[], int n){
   int i,j;
   for(i=0;i<n-1;i++){
      for(j=0;j<n-(i+1);j++){
         if (x[j]>x[j+1]){
            tukar(&x[j],&x[j+1]);
         }
      }
   }
}
```

Bubble Sort

- Pada algoritma bubble sort menggunakan nested loop 2 tingkat
- Kompleksitas $O(n^2)$
- Jika data yang akan disorting berukuran n, maka mayoritas running time akan digunakan untuk mengeksekusi statemen dalam nested loop, yaitu sebanyak n^2 kali

Time Efficiency

- **The 90/10 Rule**
  "90% of the execution time of a program is spent in executing 10% of the code "

- Most time in a program's execution is spent in a small amount of its code.

- Modifying this code is the only way to achieve any significant speedup

Algoritme #2: SELECTION SORT

- Langkah-langkah :
  - Cari nilai minimum
  - Tukar dengan posisi pertama
  - Ulangi sampai semua terurut
- Contoh
  - 64 25 12 22 11 nilai terkecil adalah 11
  - 11 25 12 22 64 nilai terkecil selanjutnya adalah 12
  - 11 12 25 22 64 dan seterusnya
  - 11 12 22 25 64
  - 11 12 22 25 64

SELECTION SORT

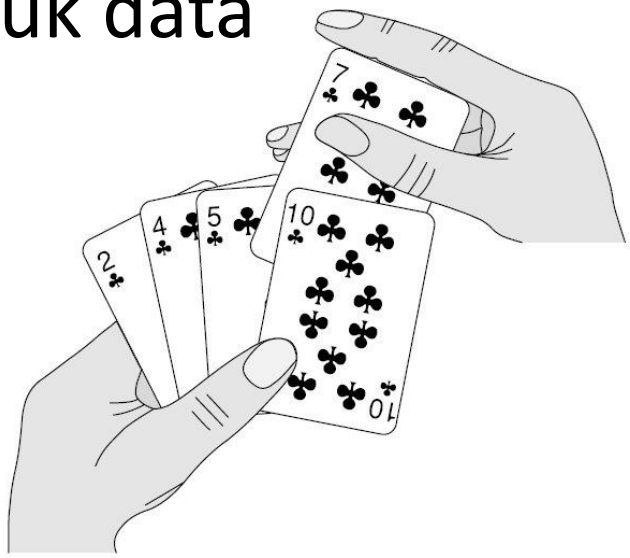
```
void selectionSort(int t[], int n) {
  int i, j;
  int min;

  for (i=0; i<n; i++) {
    min = i;
    //looping untuk cari yg terkecil
    for (j=i+1; j<n; j++) {
      if (t[j] < t[min])
        min = j;
     }
     tukar(t, i, min);
  }
}
```

SELECTION SORT

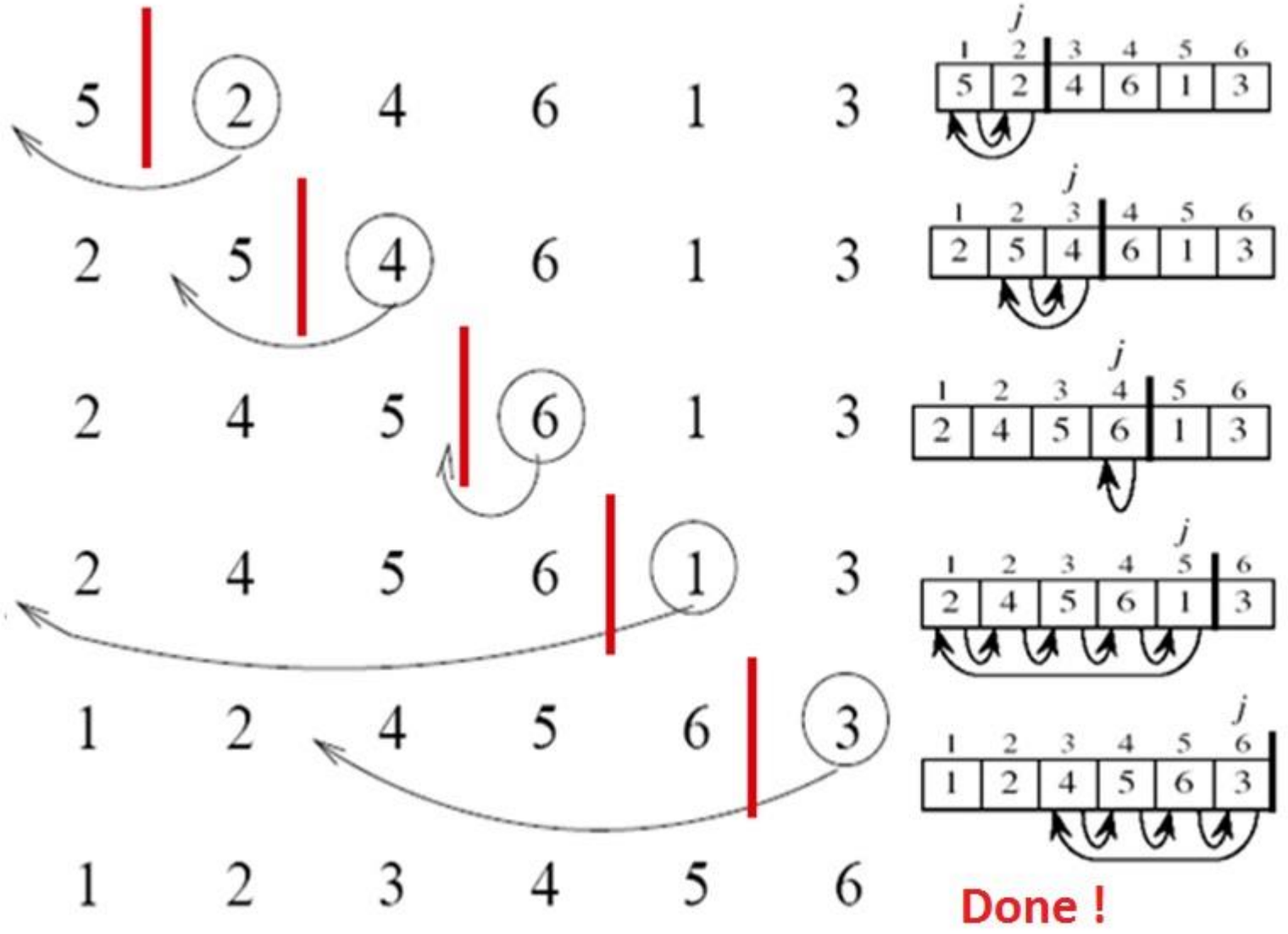
- Pada algoritma selection sort menggunakan nested loop 2 tingkat
- Kompleksitas $O(n^2)$
- Jika data yang akan disorting berukuran n, maka mayoritas running time akan digunakan untuk mengeksekusi statement dalam nested loop, yaitu sebanyak n^2 kali

Algoritme #3: INSERTION SORT

- Ide : mengurutkan kartu-kartu
- Baca elemen dalam array dari kiri ke kanan, dan tempatkan pada posisi yang sesuai
- Keuntungan: Sederhana dan mudah diimplementasikan
- Kelemahan: Kurang efisien untuk data berukuran besar



Insertion Sort Example



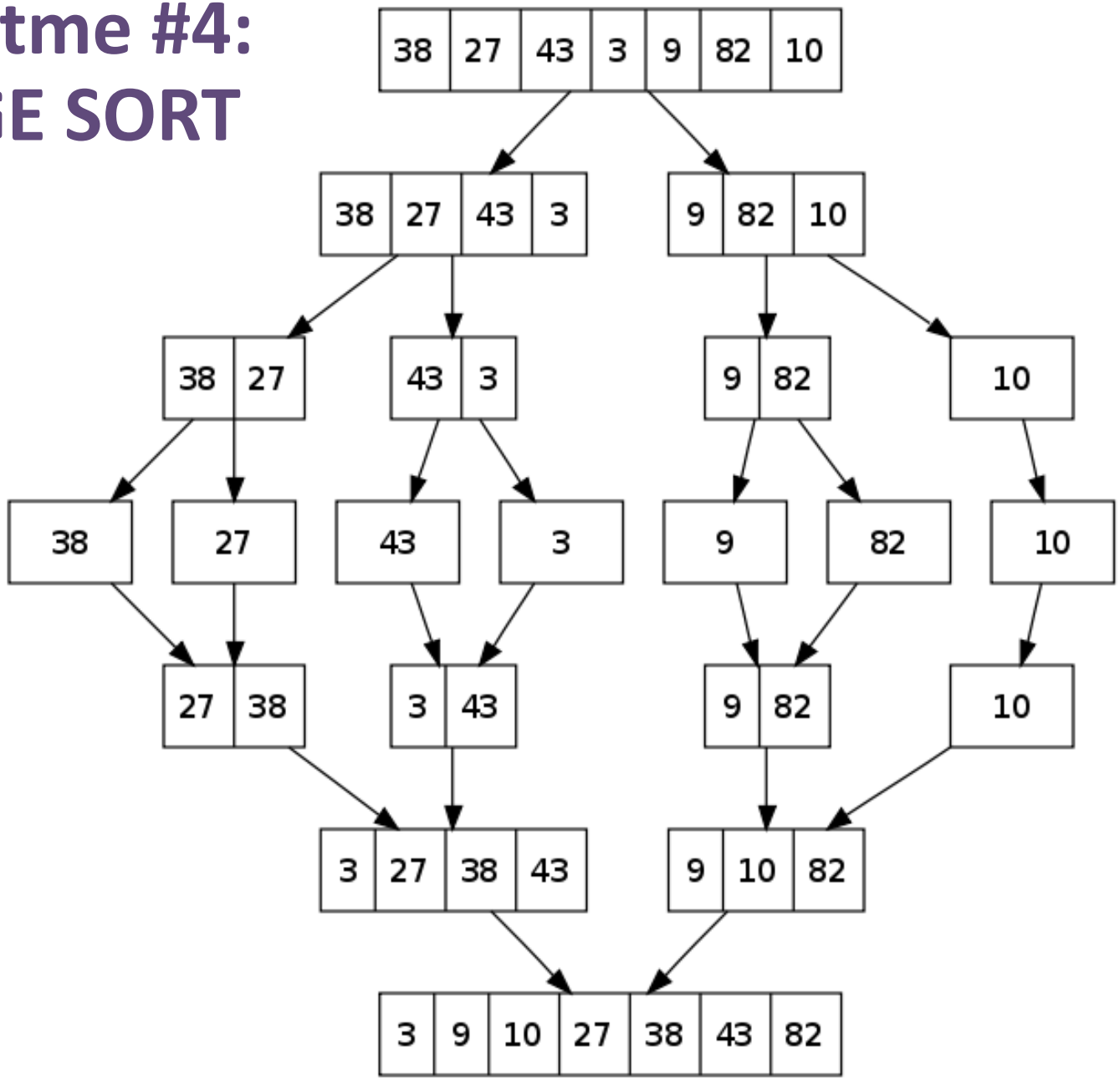
Insertion Sort Program

```
void insertionSort(int t[], int n){
  int i, j;
  index;

  for (i=1; i<n; i++) {
    index = t[i];
    j = i;
    //loop sampai mendapatkan posisi yang tepat
    while ((j>0) && (t[j-1] > index)){
      t[j] = t[j-1];
      j = j-1;
    }
    t[j] = index;
  }
}
```

Insertion Sort

- Menggunakan nested loop 2 tingkat
- Kompleksitas $O(n^2)$
- Jika inputan data yang akan disorting berukuran n, maka mayoritas running time akan digunakan untuk mengeksekusi statement dalam nested loop, yaitu sebanyak n^2 kali

Algoritme #4: MERGE SORT



Algoritme #4:
MERGE SORT: Merge

```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
```

Algoritme #4:
MERGE SORT: Merge

```
/* Merge the temp arrays back into arr[l..r]*/
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = l; // Initial index of merged subarray
while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

Algoritme #4:
MERGE SORT: Merge

```
    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Algoritme #4:
MERGE SORT: Merge Sort

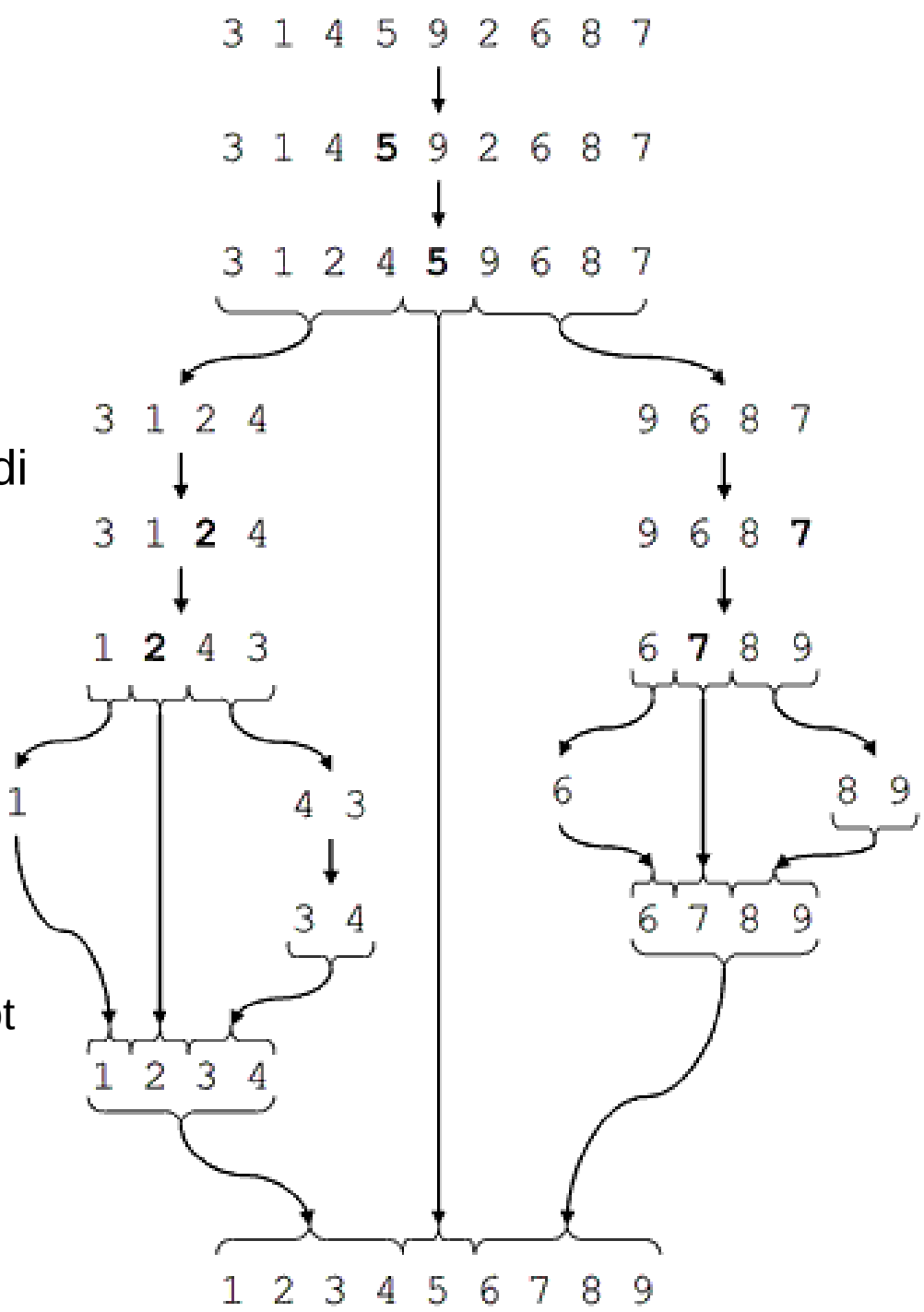
```
/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}
```

Merge Sort

- Pada algoritma ini menggunakan rekursi kemudian while looping
- Terdapat penyederhanaan masalah
- Kompleksitas O(n log n)

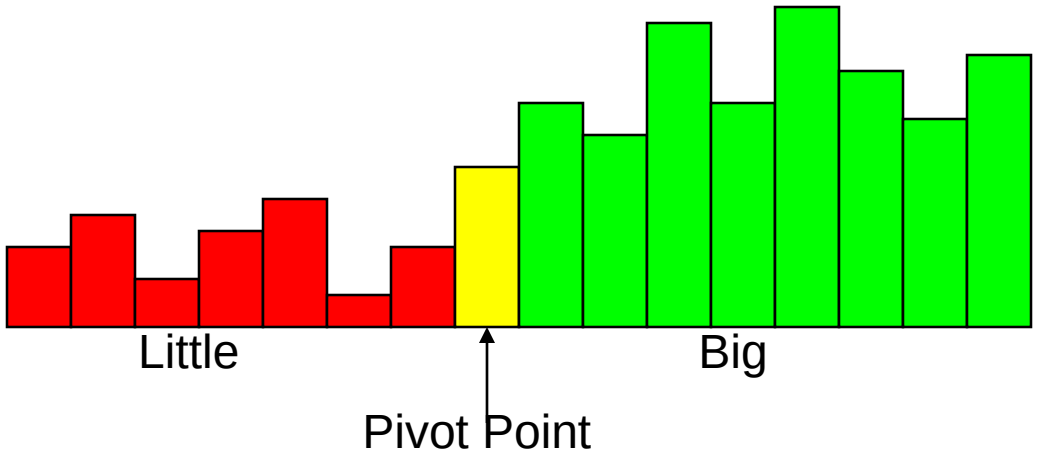
Algoritme #5: QUICKSORT

- Menerapkan strategi divide and conquer untuk membagi list menjadi dua sublist
- Tahapan
  - Ambil elemen, disebut pivot, dari list
  - Posisikan pivot sehingga elemen sebelah kiri lebih kecil dari pivot, dan elemen sebelah kanan lebih besar dari pivot
  - Lakukan hal yang sama untuk sublist sebelah kiri dan kanan pivot secara rekursif.



Quick Sort I

- Split List into “Big” and “Little” keys
- Put the Little keys first, Big keys second
- Recursively sort the Big and Little keys



Quicksort II

- Big is defined as “bigger than the pivot point”
- Little is defined as “smaller than the pivot point”
- The pivot point is chosen “at random”
- Since the list is assumed to be in random order, the last/first element of the list is chosen as the pivot point

Quicksort Split: Code

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

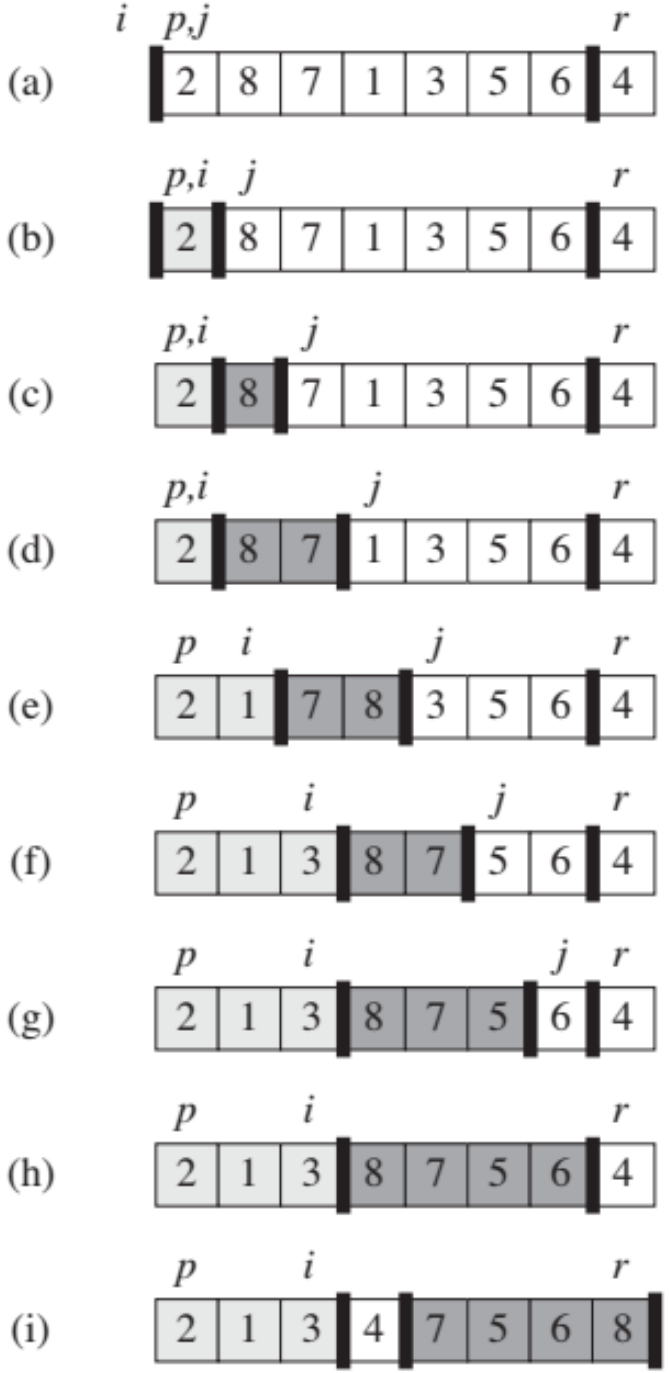
Points to last element in “Small” section.

Fixed n-1 iterations

Make “Small” section bigger and move key into it.

Else the “Big” section gets bigger.

Split Illustration



Quick Sort Running Time

- Partitioning takes O(n)
- Merging takes O(1)
- So, for each recursive call, the algorithm takes O(n)
- How many recursive calls does a quick sort need?
  - n for worst case → if pivot is least or greatest key → $O(n^2)$
  - log n for average case → O(n log n)

Studi Merge Sort dan Quick Sort

- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort
- https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Algoritme #6:
RADIX SORT

- Idea: radix sort is a sorting algorithm that sorts integers by processing individual digits
- Two classifications of radix sorts:
- Least significant digit (LSD) radix sorts
- Most significant digit (MSD) radix sorts
- LSD radix sorts process the integer representations starting from the least significant digit and move towards the most significant digit. MSD radix sorts work the other way around

Radix Sort Example
Worst case and best case = O(n)

- Original:
  - 516, 223, 323, 413, 416, 723, 813, 626, 616
- Using Queues:

First Pass:

0:
1:
2:
3: 223, 323, 413, 723,813
4:
5:
6: 516, 416, 626, 616
7:
8:

second Pass:

0:
1: 413, 813, 516, 416, 616
2: 223, 323, 723, 626
3:
4:
5:
6:
7:
8:

third Pass:

0:
1:
2: 223
3: 323
4: 413,416
5: 516
6: 616,626
7: 723
8: 813

Big-Oh to Primary Sorts

Algoritme	O
Bubble Sort	n^2
Selection Sort	n^2
Insertion Sort	n^2
Merge Sort	n log(n)
Quick Sort	n log(n)
Radix Sort	n

Hashing

HENDRA RAHMAWAN

Background

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE.

For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language.

Implementation alternatives:

- Array. O(N) for linear search or Log(N) if was sorted.
- Linked List. O(N)
- Balanced Binary Search Tree. Log(N)
- Direct Access/Addressing Table. O(1)
- Hash Table. O(1) or O(N) in worst case.

Direct-Address Tables (DAT)

A simple technique that works well when the universe U of keys is reasonably small.

- Suppose that an application needs a dynamic set in which each element has a key drawn from the universe U = {0, 1,...,m-1}, where m is not too large.
- We shall assume that no two elements have the same key.

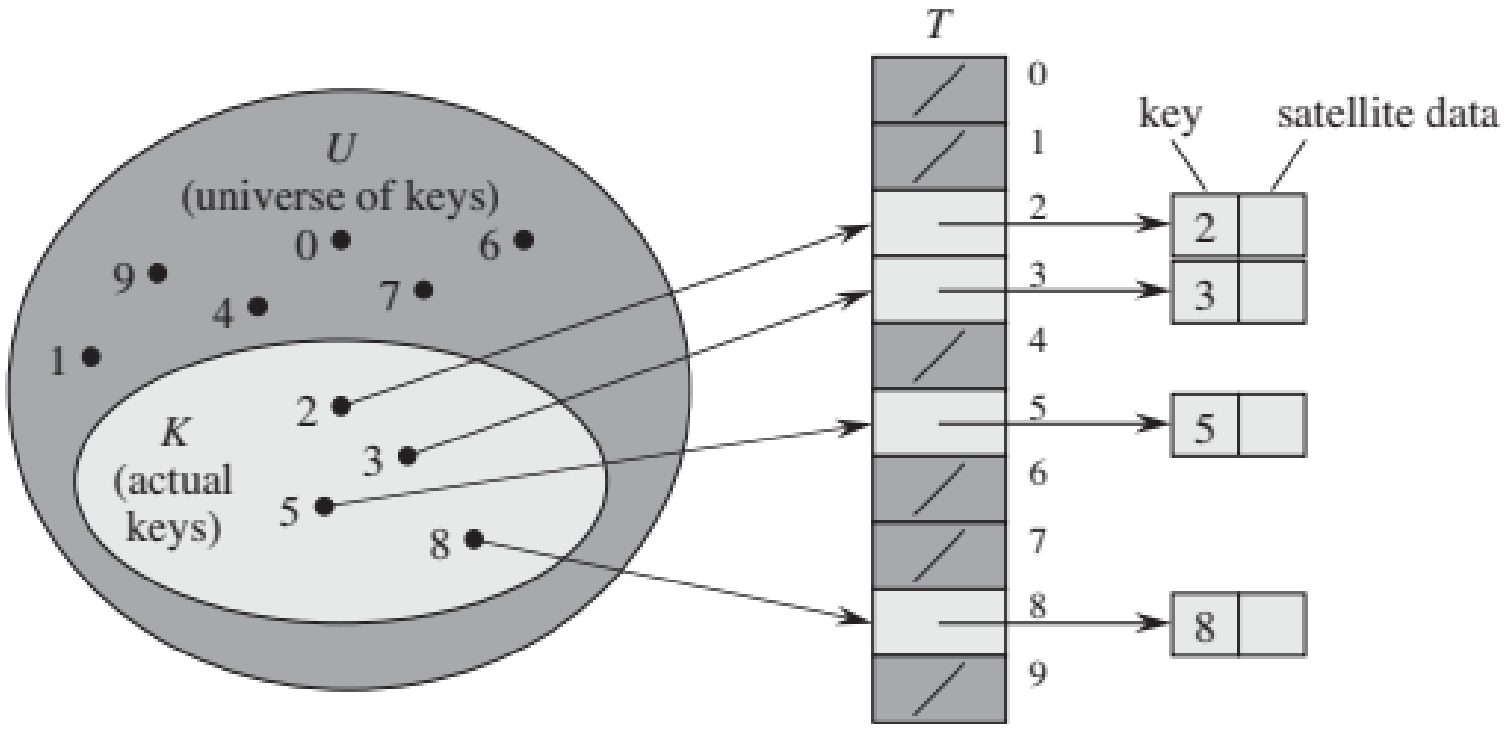
To represent the dynamic set, we use an array, or direct-address table, denoted by T[0 .. m-1]

- Eeach position, or slot, corresponds to a key in the universe U

For some applications, the direct-address table itself can hold the elements in the dynamic set.

- Store the object in the slot itself, thus saving space
- It is often unnecessary to store the key of the object, since if we have the index of an object in the table, we have its key.
- If keys are not stored, however, we must have some way to tell whether the slot is empty

Direct-Address Tables (contd.)



Direct-Address Tables (contd.)

DAT Algorithms:

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] =$ NIL

The downside of direct addressing is obvious:

- If the universe U is large, storing a table T of size |U| may be impractical, or even impossible, given the memory available on a typical computer.
- Furthermore, the set K of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted.

Hash Tables

A **hash table** is an effective data structure for implementing **dictionaries**.

- Although searching for an element in a hash table can take as long as searching for an element in a linked list—$\Theta(n)$ time in the worst case—in practice, hashing performs extremely well.
- Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct address table.

- we can **reduce the storage requirement** to $\Theta(|K|)$ while we maintain the benefit that searching for an element in the hash table still requires only $O(1)$ time.
- This bound is for **the average-case time**, whereas for direct addressing it holds for the worst-case time.

Hash Tables (contd.)

With direct addressing, an element with key k is stored in slot k.

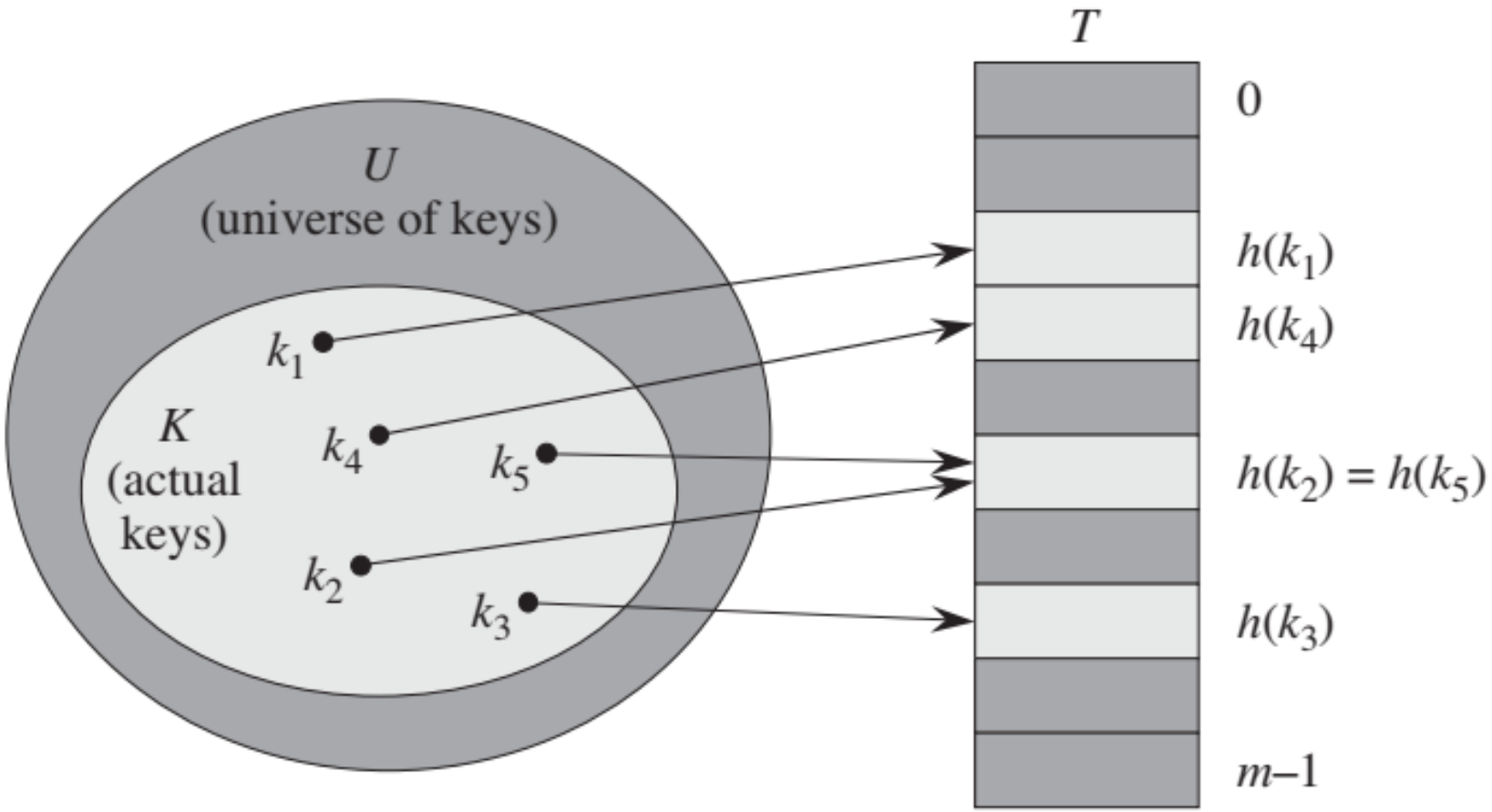
With hashing, an element is stored in slot **h(k)**.

- We use a hash function h to compute the slot from the key k.
- h maps the universe U of keys into the slots of a hash table T[0..m-1]
- h: U → {0, 1, ..., m-1}

The size m of the hash table is typically much less than |U|

- We say that an element with key k hashes to slot h(k)
- We also say that h(k) is the hash value of key k
- The hash function **reduces** the range of array indices and hence the size of the array.
- Instead of a size of |U|, the array can have size m.

Hash Tables (contd.)



Hash Tables: Collisions

Collision: two keys may hash to the same slot.

- We have effective techniques for resolving the conflict created by collisions.

The ideal solution would be to avoid collisions altogether.

- One idea is to make h appear to be “random,” thus avoiding collisions or at least minimizing their number.

Because |U| > m, however, there must be at least two keys that have the same hash value

- Avoiding collisions altogether is therefore impossible
- We still need a method for resolving the collisions that do occur.

Collision resolution:

- **Chaining**
- **Open address**

Collision Resolution By Chaining

The simplest collision resolution technique

Place all the elements that **hash to the same slot** into the **same linked list**

The dictionary operations on a hash table T are easy to implement when collisions are resolved by chaining:

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

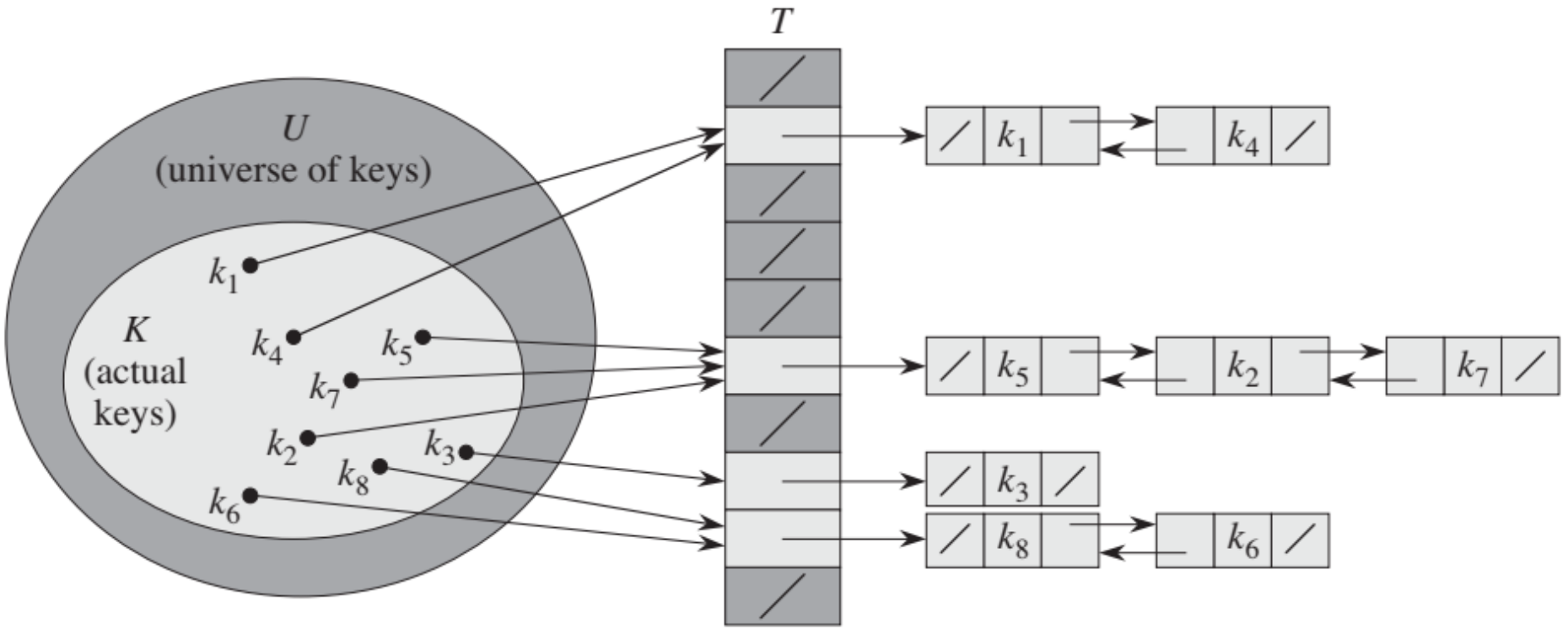
CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

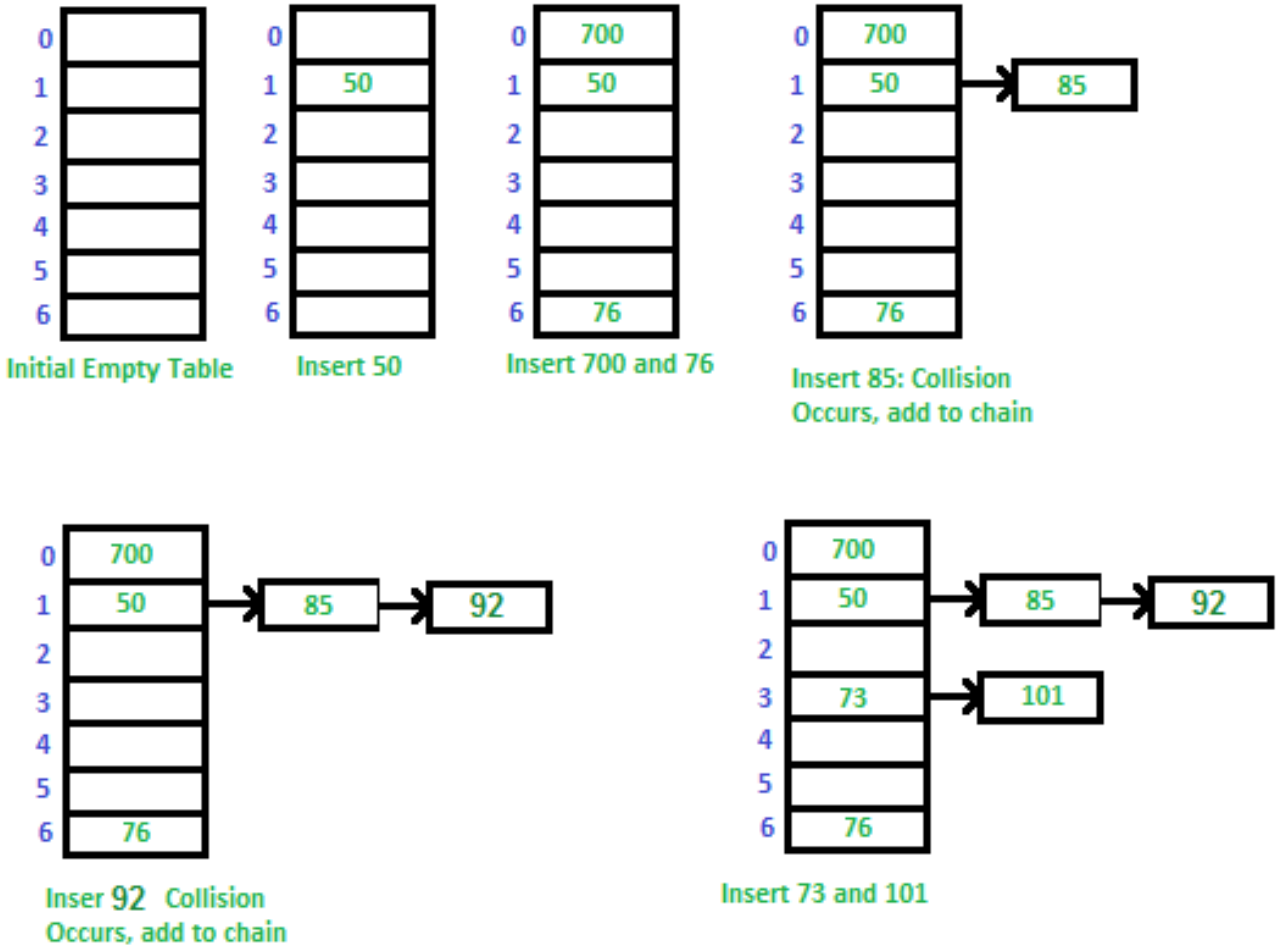
1 delete x from the list $T[h(x.key)]$

Collision Resolution by Chaining: contd.



Collision Resolution by Chaining: Example

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Hash Functions

A good hash function satisfies:

- Each key is equally likely to hash to any of the m slots, but we rarely know the probability distribution from which the keys are drawn
- Is independent of where any other key has hashed to, but the keys might not be drawn independently

Occasionally we do know the distribution.

- For example, if we know that the keys are random real numbers k independently and uniformly distributed in the range $0 \leq k < 1$, then the hash function $h(k) = \lfloor km \rfloor$ satisfies the condition of simple uniform hashing.

Interpreting Keys as Natural Number

Most hash functions assume that the universe of keys is the set N = {1, 2, ...} of natural numbers.

If the keys are not natural numbers, interpret them as natural numbers.

For example, we can interpret a character string **pt** as an integer expressed in suitable radix notation.

- interpret the identifier **pt** as the pair of decimal integers (112, 116), since p = 112 and t = 116 in the ASCII character set.
- express as a radix-128 integer. **pt** becomes (112*128) + 116 = 14452.

Hash Functions: The Division Method

Map a key k into one of m slots by taking the **remainder of k divided by m**.

The hash function is **h(k) = k mod m**.

For example, if the hash table has size m =12 and the key is k = 100, then h(k) = 4.

We usually avoid certain values of m.

- For example, m should not be a power of 2, since if $m = 2^p$, then h(k) is just the p lowest-order bits of k.
- Unless we know that all low-order p-bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key.

A prime not too close to an exact power of 2 is often a good choice for m.

Hash Function: The Division Method (contd.)

For example, suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly n = 2000 character strings, where a character has 8 bits.

We don't mind examining an average of 3 elements in an unsuccessful search.

We allocate a hash table of size m = 701.

We could choose m = 701 because it is a prime near 2000/3 but not near any power of 2.

Treating each key k as an integer, our hash function would be h(k) = k mod m.

Hash Function: Multiplication Method

Operates in two steps:

1. Multiply the key k by a constant A in the range 0 < A < 1 and extract the fractional part of kA.
2. Multiply this value by m and take the floor of the result.

In short the hash function is: $h(k) = \lfloor m\ (k\ A\ mod\ 1) \rfloor$

$kA\ mod\ 1$ means the fractional part of kA, that is, $kA\ -\ \lfloor kA \rfloor$. Thus: $h(k) = \lfloor m\ (kA\ -\ \lfloor kA \rfloor) \rfloor$

An **advantage** of the multiplication method is that the **value of m is not critical**.

Typically choose m to be a power of 2 ($m = 2^p$ for some integer p).

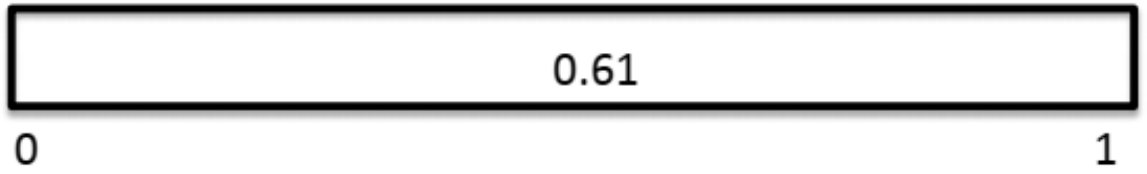
Knuth suggests: $A \approx \frac{(\sqrt{5}-1)}{2} = 0.6180339887$

Multiplication Method: A Simple Example

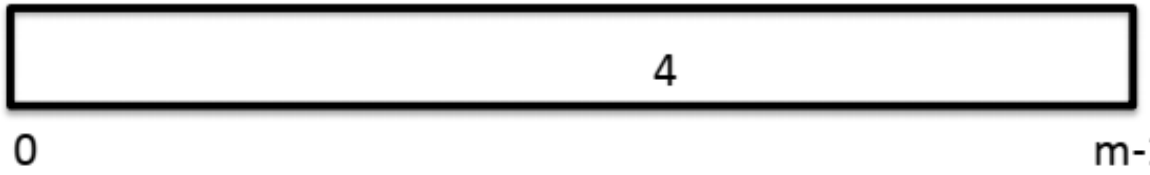
Key k = 1; m = 8 slots

(1) A = .61

(2) K A = 1 x .61 = .61



(3) Floor (f m) = Floor(.61 x 8) = Floor(4.8) = 4

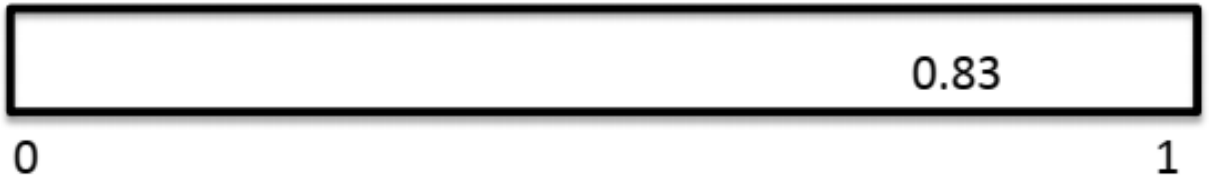


Multiplication Method: A Simple Example

Key k = 3; m = 8 slots

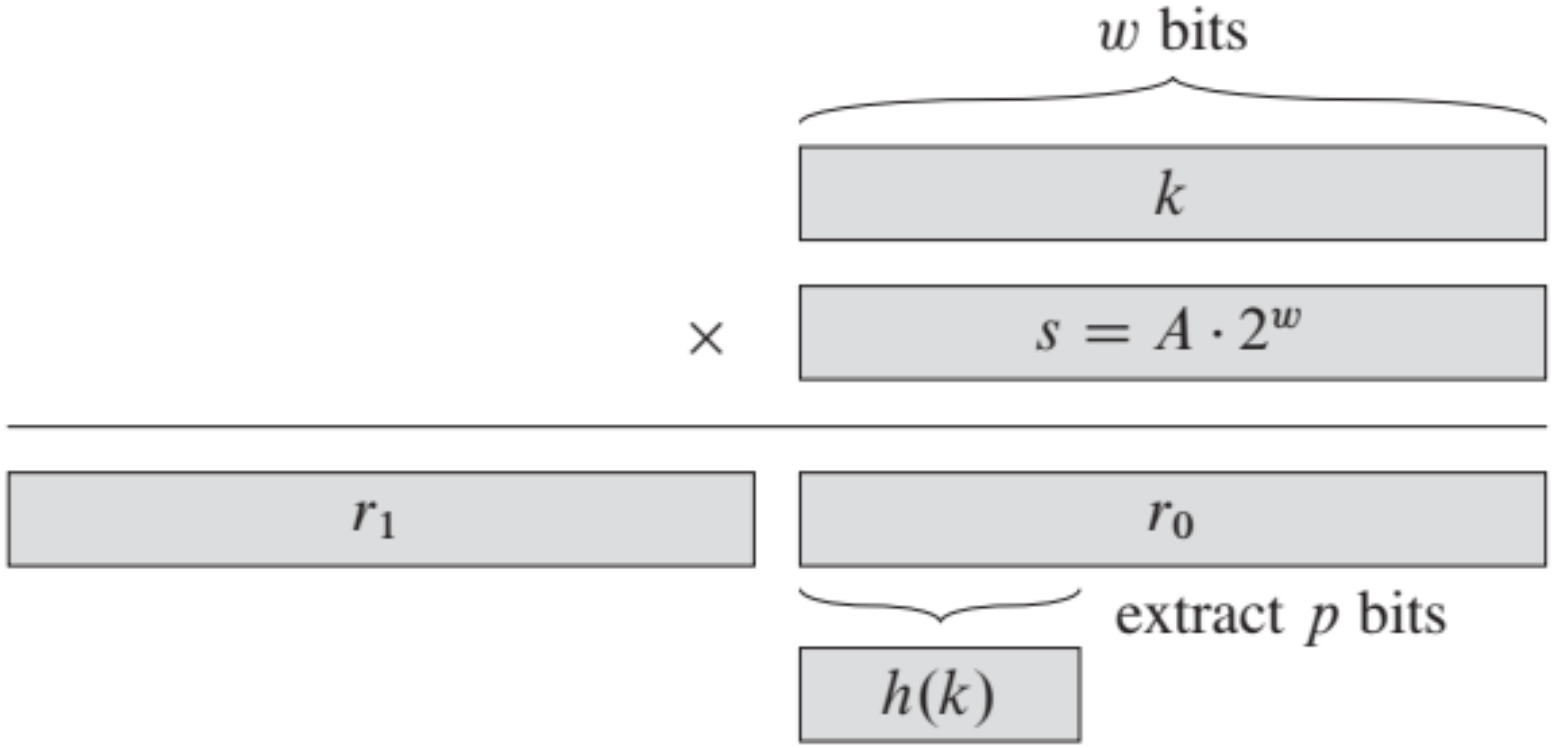
(1) A = .61

(2) kA = 3 x .61 = 1.83



(3) Floor (f m) = Floor(.83 x 8) = Floor(6.64) = 6

Multiplication Method: Computer Implementation



Collision Resolution by Open Addressing

All elements occupy the hash table itself.

Each table entry contains:

- an element of the dynamic set, or
- NIL.

When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.

No lists and no elements are stored outside the table, unlike in chaining -> closed hash.

The hash table can “fill up” so that no further insertions can be made.

The advantage of open addressing is that it avoids pointers altogether.

Instead of following pointers, we compute the sequence of slots to be examined

Collision Resolution by Open Addressing (contd.)

To perform insertion using open addressing, we successively examine, or probe, the hash table until we find an empty slot in which to put the key.

To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input.

Thus, the hash function becomes:

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$$

With open addressing, we require that for every key k, the probe sequence

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$$

Open Addressing: Hash-Insert

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == NIL
5            T[j] = k
6            return j
7       else i = i + 1
8  until i == m
9  error "hash table overflow"
```

In the pseudocode, we assume that the elements in the hash table T are keys with no satellite information; the key k is identical to the element containing key k.

Each slot contains either a key or NIL (if the slot is empty)

Open Addressing: Hash-Search

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted.

Therefore, the search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence.

Open Addressing: Hash-Delete

Deletion from an open-address hash table is difficult.

When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it.

If we did, we might be unable to retrieve any key k during whose insertion we had probed slot i and found it occupied.

We can solve this problem by marking the slot, storing in it the special value DELETED instead of NIL.

We would then modify the procedure HASH-INSERT to treat such a slot as if it were empty so that we can insert a new key there.

We do not need to modify HASH-SEARCH, since it will pass over DELETED values while searching.

Open Addressing: Linear Probing

Uses the hash function:

$$h(k,i) = (h'(k) + i) \bmod m$$

Given key k, we first probe T[h'(k)], i.e., the slot given by the auxiliary hash function. We next probe slot T[h'(k)+1], and so on up to slot T[m-1]. Then we wrap around to slots T[0], T[1], ... until we finally probe slot T[h'(k)-1].

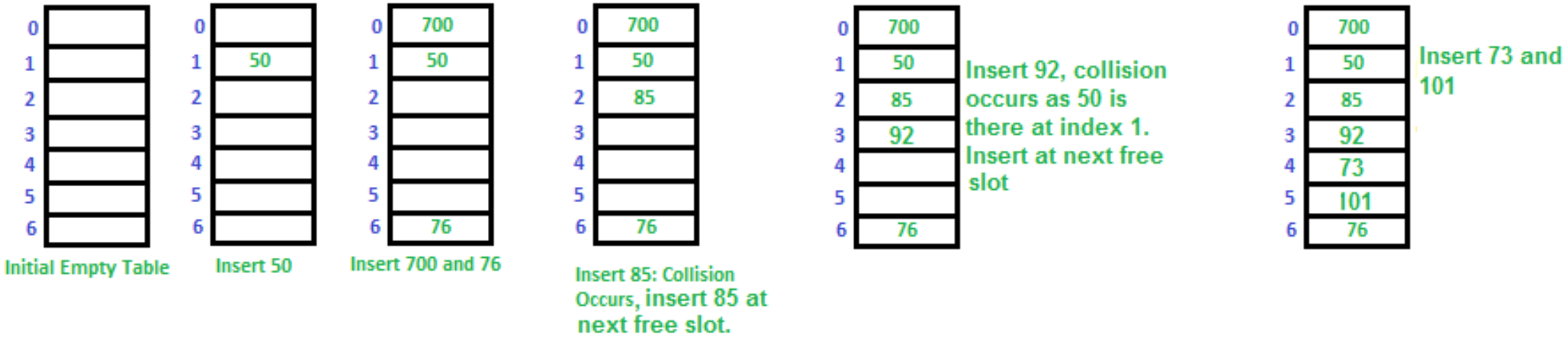
Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.

Linear probing is easy to implement, but it suffers from a problem known as **primary clustering**.

- increasing the average search time

Open Addressing: Linear Probing example

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Open Addressing: Quadratic Probing

Uses a hash function of the form:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

The initial position probed is T[h’(k)]

Later positions probed are offset by amounts that depend in a quadratic manner on the probe number i.

This method works much better than linear probing, but to make full use of the hash table, the values of c1, c2, and m are constrained.

Open Addressing: Double Hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

Double hashing uses a hash function of the form:

$$h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$$

Both h1 and h2 are auxiliary hash functions

The initial probe goes to position $T[h_1(k)]$

Successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m.

Open Addressing: Double Hashing Example

$$h_1(k) = k \bmod 11; h_2(k) = k \bmod 7 + 1$$

Keys to insert: 14; 17; 25; 3

- 14 goes to slot 14mod11 = 3; 17 goes to slot 17mod11=6;
- 25 initially results in collision for first probe and so goes to slot (25mod11+(25mod7+1))mod11 = 8 mod 11 = 8;
- 3 has two collisions and eventually is placed in (3mod11+(2 x (3mod7 + 1))mod11=(3+8)mod 11 =11mod11=0)

Open Addressing: Double Hashing Example

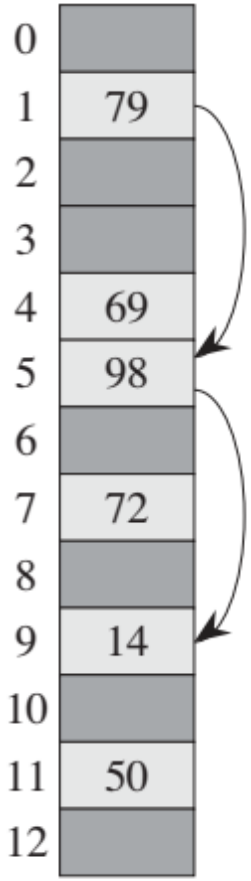
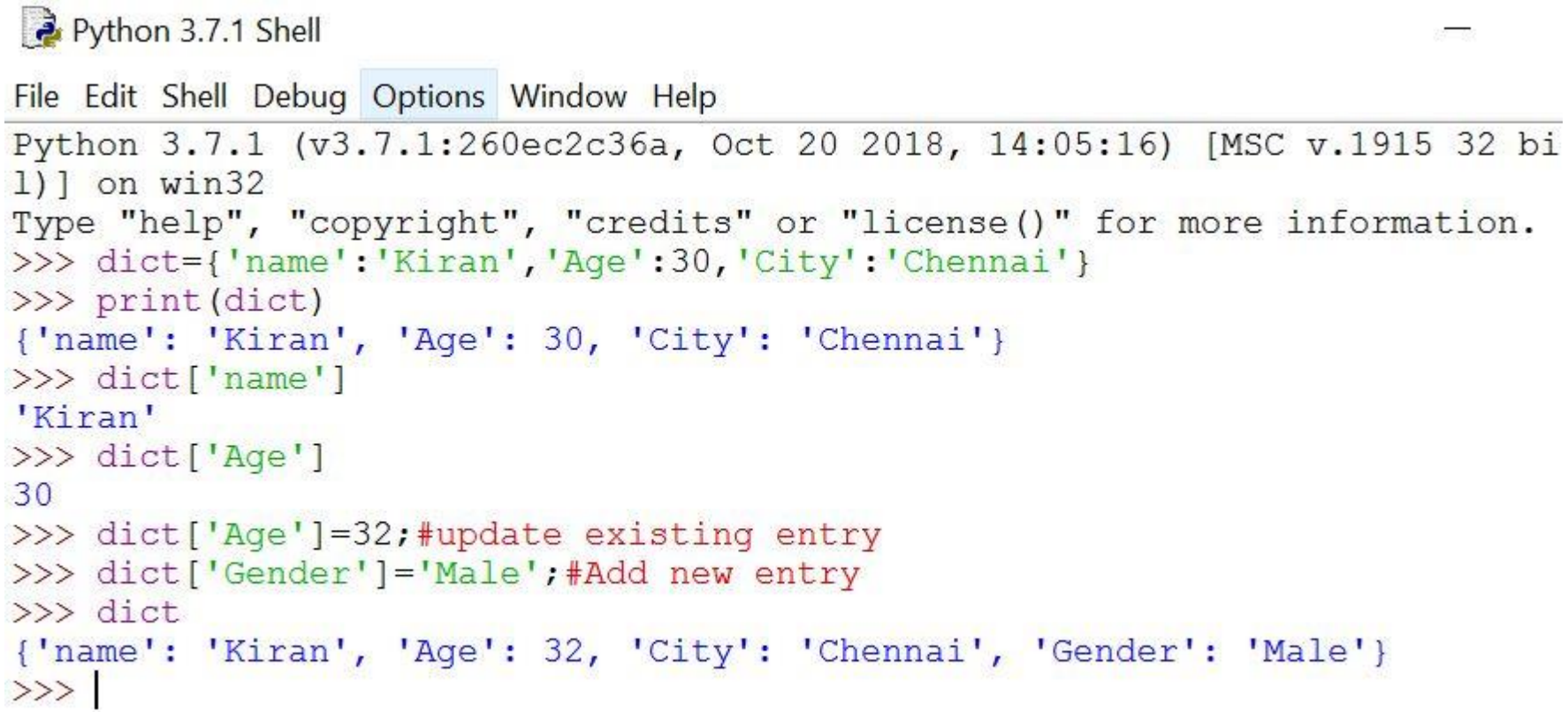


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, we insert the key 14 into empty slot 9, after examining slots 1 and 5 and finding them to be occupied.

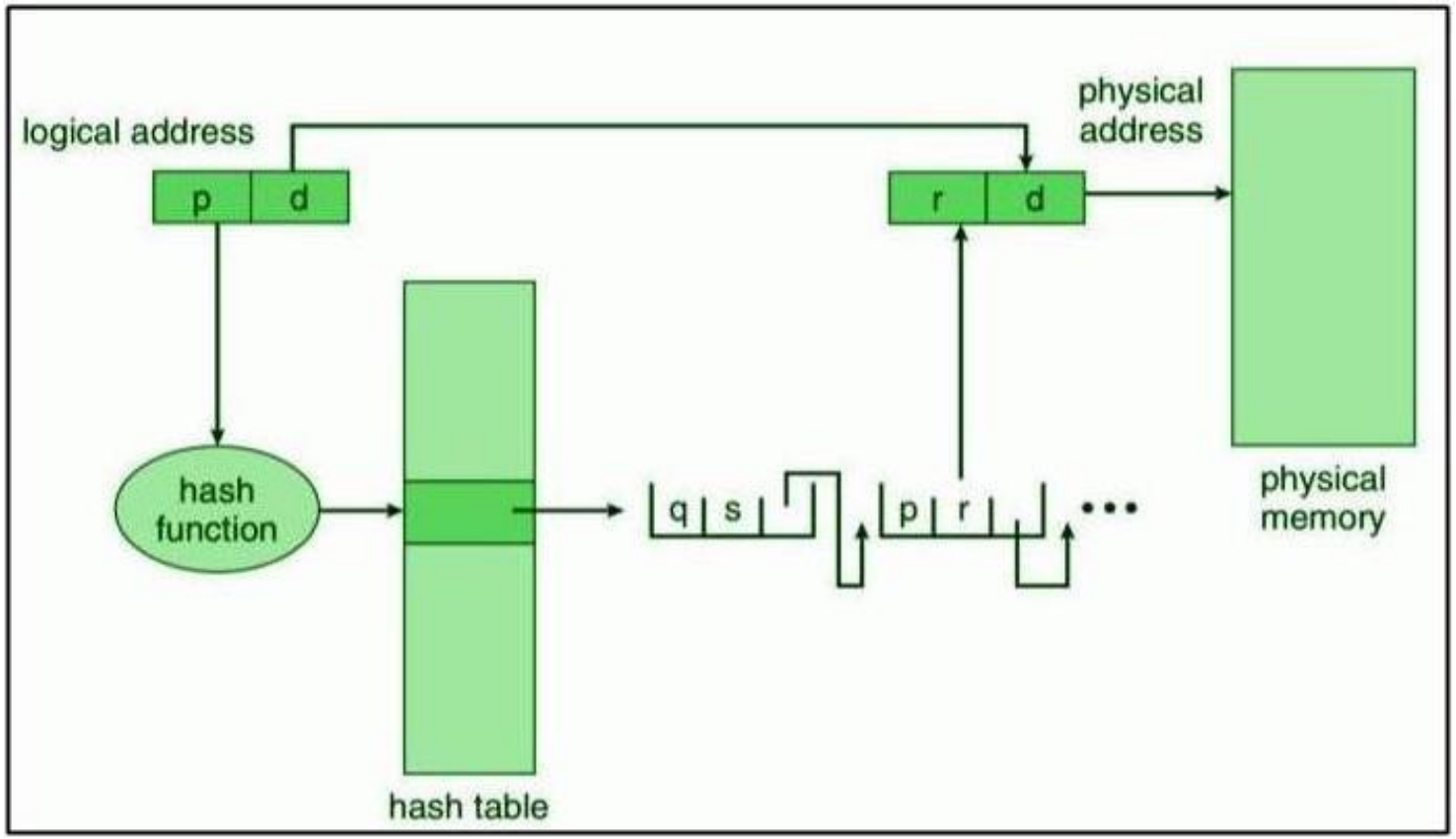
Dictionary in C++: map

```cpp
#include <iostream>
#include <map>
#include <string>

int main()
{
    //create a map that stores strings indexed by strings
    std::map<std::string, std::string> m;

    //add some items to the map
    m["cat"] = "mieow";
    m["dog"] = "woof";
    m["horse"] = "neigh";
    m["fish"] = "bubble";

    //now loop through all of the key-value pairs
    //in the map and print them out
    for ( auto item : m )
    {
        //item.first is the key
        std::cout << item.first << " goes ";

        //item.second is the value
        std::cout << item.second << std::endl;
    }

    //finally, look up the sound of a cat
    std::cout << "What is the sound of a cat? " << m["cat"]
              << std::endl;

    return 0;
}
```

Dictionary in Python



Page Table


logical address
p
d
physical address
r
d
hash function
q
s
p
r
physical memory
hash table

Hash Table Visualization

https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html

https://www.cs.usfca.edu/~galles/visualization/OpenHash.html

https://visualgo.net/en/hashtable

Reference

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2009. Introduction to algorithms. MIT press.

http://117.247.251.79:8080/jspui/bitstream/1/1348/1/t._cormen_-_introduction_to_algorithms_3rd_edition.pdf