Data Structure for Language Processing

Bhargavi H. Goswami
Assistant Professor
Sunshine Group of Institutions

INTRODUCTION:

- Which operation is frequently used by a Language Processor?
- Ans: Search.
- This makes the design of data structures a crucial issue in language processing activities.
- In this chapter we shall discuss the data structure requirements of LP and suggest efficient data structure to meet there requirements.

Criteria for Classification of Data Structure of LP:

- 1. Nature of Data Structure: whether a “linear” or “non linear”.
- 2. Purpose of Data Structure: whether a “search” DS or an “allocation” DS.
- 3. Lifetime of a data structure: whether used during language processing or during target program execution.

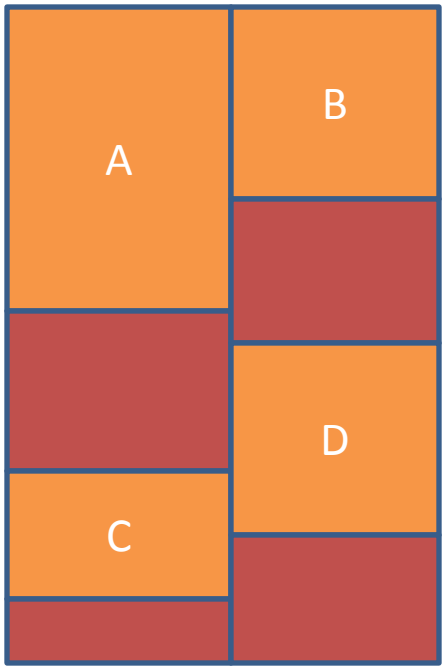
Linear DS

- Linear data structure consist of a linear arrangement of elements in the memory.
- Advantage: Facilitates Efficient Search.
- Dis-Advantage: Require a contagious area of memory.
- Do u consider it a problem? Yes or No?
- What the problem is?
- Size of a data structure is difficult to predict.
- So designer is forced to overestimate the memory requirements of a linear DS to ensure that it does not outgrow the allocated memory.
- Disadvantage: Wastage Of Memory.

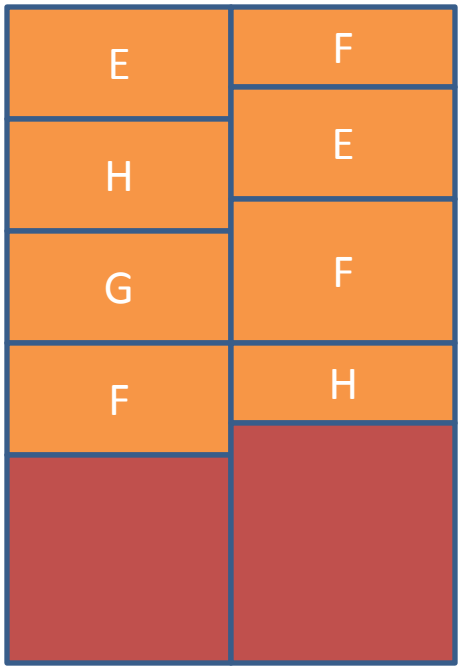
Non Linear DS

- Overcomes the disadvantage of Linear DS. HOW?
- Elements of Non Linear DS are accessed using pointers.
- Hence the elements need not occupy contiguous areas of memory.
- Disadvantage: Non Linear DS leads to lower search efficiency.

Linear & Non-Linear DS



Linear



Non Linear

Search Data Structures

- Search DS are used during LP'ing to maintain attribute information concerning different entities in source program.
- Fact: The entry for any entity is created only once but may be searched for large number of times.
- Search efficiency is therefore very important.

Allocation Data Structures

- Fact: The address of memory area allocated to an entity is known to the user(s) of that entity.
- Means, no search operations are conducted on them.
- So what remains the important criteria for allocation data structures?
  - Speed of allocation and deallocation
  - Efficiency of memory utilization

Use of Search and Allocation DS:

- LP uses both search DS and allocation DS during its operation.
- Use of Search DS: To constitute various tables of information.
- Use of Allocation DS: To handle programs with nested structures of some kind.
- Target program rarely uses search DS.

e.g Consider Following Pascal Program:

```
Program Sample(input,output);
   var
        x,y : real;
        i     : integer;
   Procedure calc(var a,b : real);
       var
            sum : real;
       begin
            sum := a+b;
            ---
      end calc;
   begin {Main Program}
   ----
end.
```

- The definition of procedure ‘calc’ is nested inside the main program.
- Symbol tables need to be created for the main program as well as for procedure ‘calc’.
- We call them $Symtab_{sample}$ and $Symtab_{calc.}$
- What DS these symbol tables are?

  Search or Allocation?
- Ans: Search Data Structures. Y?
- During compilation, the attributes of a symbol are obtained by searching appropriate symbol table.

- Now, memory needs to be allocated to $Symtab_{sample}$ and $Symtab_{calc}$
- How would we do it?
- Ans: Using an Allocation DS.
- The addresses of these tables are noted in a suitable manner.
- Hence no searches are involved in locating $Symtab_{sample}$ and $Symtab_{calc.}$

e.g Consider Following Pascal and C segments:

- Pascal:

```
var p : ↑ integer;
begin
    new (p);
```

- C:

```
float *ptr;
ptr = (float*)calloc(5,sizeof(float));
```

- The Pascal call new(p): allocates sufficient memory to hold an integer value and puts the address of this memory area in p.
- The C statement ptr=… : allocates a memory area sufficient to hold 5 float values and puts its address in ptr.
- Means, access to these memory area are implemented through pointers. i.e p and ptr.
- Conclusion: No search is involved in accessing the allocated memory.

SEARCH DATA STRUCTURES

Search Data Structures Topic List

- Entry Formats
- Fixed and variable length entries
- Hybrid entry formats
- Operations on search structures
- Generic Search Procedures
- Table organizations
- Sequential Search Org
- Binary Search Org
- Hash table Org
- Hashing functions
- Collision handling methods.
- Linked list
- Tree Structured Org

Search Data Structure:

- When we talk of ‘Search’ what is the basic requirement?
- Ans. ‘Key’. Key is the symbol field containing name of an entity.
- Search Data Structure (also called search structure) is a set of entries, each entry accommodating the information concerning one entity.
- Each entry in search structure is a set of fields i.e a record, a row.
- Each entry is divided into two parts:
  - Fixed Part
  - Variant Part
- The value in fixed (tag) part determines the information to be stored in the variant part of the entry.

Entries in the symbol table of a compiler have following field:

Tag Value	Variant Part Fields
Variable	type, length, dimension info
Procedure	address of parameter list, number of parameters
Function	type of returned value, length of returned value, address of address of parameter list, number of parameters
Label	statement number

Fixed Length Entry:

1. Symbol
2. Class
3. Type
4. Length
5. Dimension Information
6. Parameter List Address
7. No. of Parameters
8. Type of returned value
9. Length of returned value
10. Statement number.

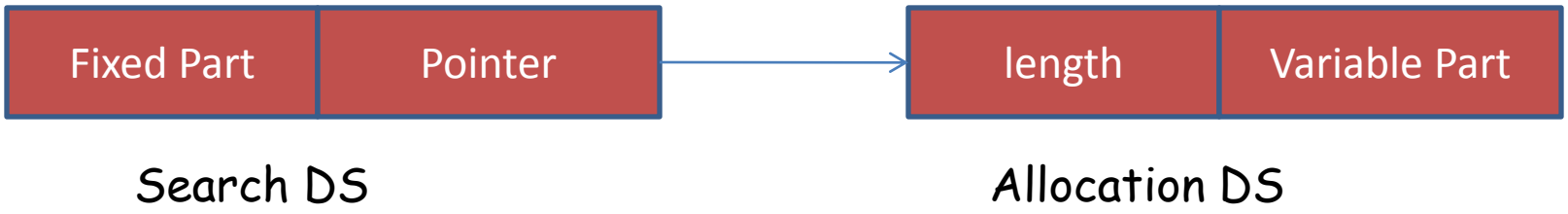
Variable Length Entry:

- 1. Name
- 2. Class
- 3. Statement Number
- When class = label, all fields excepting name, class and statement number are redundant.
- Here, Search method may require knowledge of length of entry.
- So the record would contain following fields:
  - 1. A length field
  - 2. Fields in fixed part including tag field
  - 3. $\{ f_j \mid f_j \in SF_{Vj} \text{ if tag} = V_j \}$

Fixed v/s Variable

- For each value V_i in the tag field, the variant part of the entry consists of the set of fields SF_{Vi}.
- Fixed Length Entry Format:
  - 1. Fields in the fixed part of the entry.
  - 2. U_{vi} SF_{vi}, i.e the set of fields in all variant parts of the entry.
- In fixed length entries, all the records in search structure have an identical format.
- This enables the use of homogeneous linear data structures like arrays.
- Drawback?
- Inefficient use of memory. How?
- Many records may contain redundant fields.
- Solution?
- Variable Length Entry Format:
  - Fields in the fixed part of entry, including the tag field
  - $\{ f_j \mid f_j \in SF_{Vj}$ if tag = $V_j \}$
- This entry format leads to compact organization in which no memory wastage occurs.

Hybrid Entry Formats:



- Compromise between Fixed and Variable entry formats.
- Why this is needed?
- To combine access efficiency of Fixed Entry Format with memory efficiency of Variable Entry Format.
- What is done here?
- Each entry is divided into two halves

  i.e Fixed Part and Variable Part
- Data Structure:
  - Fixed Part : Search DS/ Linear DS. Y? Require Efficient Searching.
  - Variable Part : Allocation DS/ Linear / Non Linear DS. Y? Fixed part has pointer field which do not need searching in variable part.

Operations on Search Structures:

- 1. Operation add: Add the entry of a symbol. Entry of symbol is created only once.
- 2. Operation search: Search and locate the entry of a symbol. Searching may be performed for more than once.
- 3. Operation delete: Delete the entry of a symbol. Uncommon operation.

Algorithm: Generic Search Procedure:

- 1. Make a *prediction* concerning the entry of search data structure which symbol s may be occupying. We call this entry e.
- 2. Let s_e be the symbol occupying e^{th} entry. *Compare* s with s_e. Exit with success if the two match.
- 3. Repeat step 1 and 2 till it can be concluded that the symbol *does not exist* in the search data structure.

- Each comparison of step 2 is called a probe.
- How to determine the efficiency of search procedure?
- Ans: No. of probes performed by search procedure.
- Probe Notations:
  - Ps : Number of probes in successful search
  - Pu : Number of probes in an unsuccessful search.

Table Organization:

- Entries of table occupy adjoining areas of memory. Adjoining areas here means ‘previous entry’ and ‘next entry’.
- **Positional Determinacy:** Tables using fixed length entry organization possess this property. This property states that the address of an entry in a table can be determined from its entry number.
- Eg: Address of the e^{th} entry is

$$a + (e - 1).\ L$$

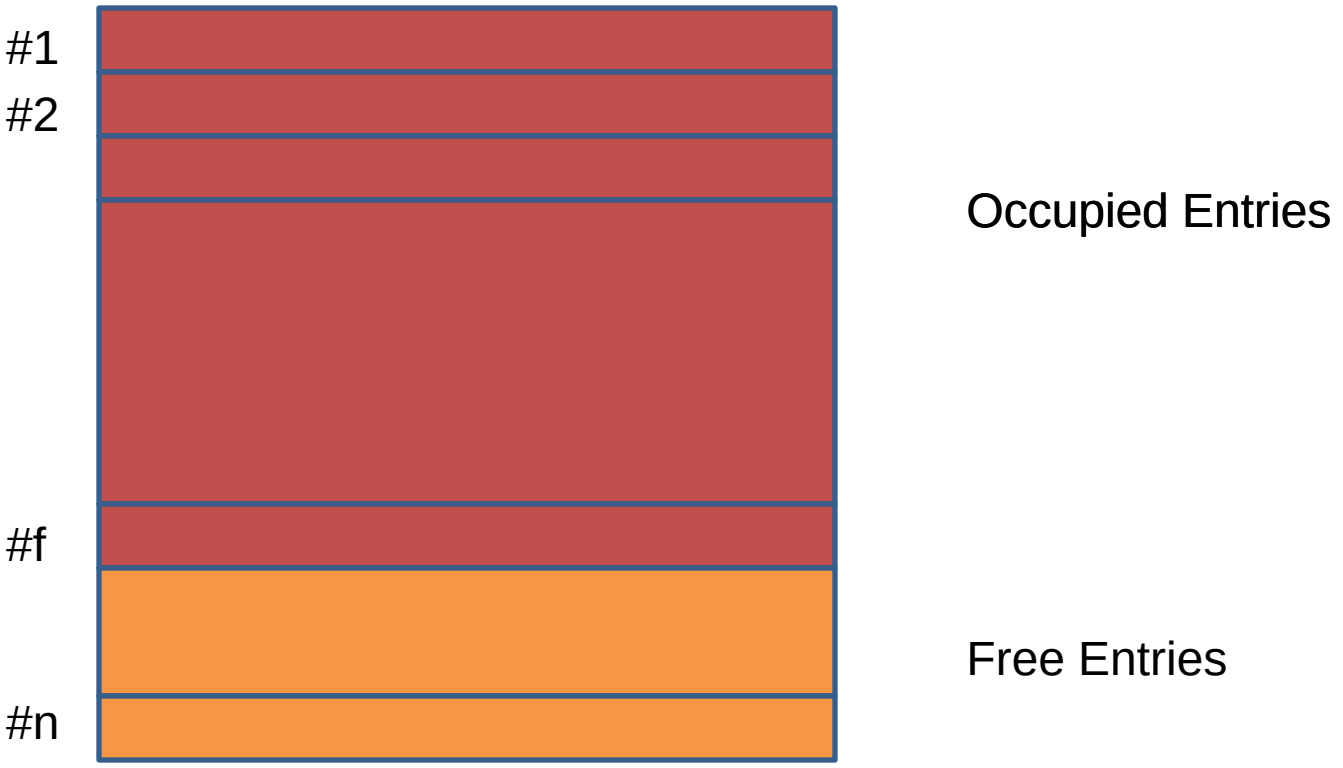
  a : address of first entry.

  L : length of an entry.

  e : entry number.

- Use of Positional Determinacy:
  - Representation of symbols by e
  - Entry number in the search structure
  - Intermediate code generated by LP

A Typical State of a Table Using Sequential Search Organization



n: Number of entries in the table
f: Number of occupied entries

Sequential Search Organization

- Search for a symbol: All active entries in the table have the same probability of being accessed.
- Ps = f/2 for a successful search
- Pu = f for an unsuccessful search
- Following an unsuccessful search, a symbol may be entered in the table using an add operation.
- **Add a symbol**: The symbol is added to the first free entry in the table. The value of f is updated accordingly.
- Delete a symbol: Two ways:
  - **Physical Deletion**: an entry is deleted by erasing or by overwriting. If the dth entry is to be deleted, entries d+1 to f can be shifted 'up' by one entry each. This would require (f-d) shift operations in symbol table. Efficient alternate would be to move fth entry into dth position, requiring only one shift operation.
  - **Logical Deletion**: is performed by adding some information to the entry to indicate its deletion. How to implement it? By introducing a field to indicate whether an entry is active or deleted.

Binary Search Organization

- All entries in a table are assumed to satisfy an ordering relation.
- '<' relation implies that the symbol occupying an entry is 'smaller than' the symbol occupying the next entry.

Algorithm 2.2 (Binary Search)

- 1. start:=1; end:=f;
- 2. while start <= end
  - (a) e:= [(start+end)/2]; where [] implies a rounded quotient. Exit with success if $s=s_e$.
  - (b) if $s<s_e$ then end:=e-1; else start:=e+1;
- 3. Exit with failure.

- For a table containing f entries we have $p_s <= [\log_2 f]$ and $p_u=[\log_2 f]$.
- What is the problem with this search organization?

- Ans: The requirement that the entry number of a symbol in the table should not change after an add operation. Y?
- Because its used in IC.
- Thus, this forbids both, addition and deletion during language processing.
- Hence, binary search organization is suitable only for a table containing a fixed set of symbols.

Hash Table Organization:

- There are three possibilities exist concerning the predicted entry
  - Entry may be occupied by s
  - Entry may be occupied by some other symbol
  - Entry may be empty.
- Which of the above possibility is called a collision?
- Ans: second case, i.e s != s_e
- Algorithm 2.3 (Hash Table Management)
  - 1. e:=h(s);
  - Exit with success if s = s_e and with failure if entry e is unoccupied.
  - Repeat steps 1 and 2 with different functions h', h'',etc.

- n : number of entries in the table
- f : number of occupied entries in the table
- P : Occupation density in table, i.e f/n
- k : number of distinct symbols in source language
- k_p: number of symbols used in some source program
- S_p: set of symbols used in some source program
- N : Address space of the table.
- K : Key space of the system
- K_p: Key space of a program

- Hashing function has the property:

  $1 <= h(symb) <= n$
- Direct Entry Organization:
  - If k<=n we can select ‘one to one’ function as hashing function h.
  - This will eliminate collision.
  - Will require large symbol table.
  - Better solution K_p => N which is nearly one to one for set of symbols S_p.
- Effectiveness of a hashing organization depends on average value of p_s.
- If k_p increases, p_s should also increase.
- Assignment Question: What is folding?
- Assignment Question: Write a Short Note on Hashing function.

Collision Handling Methods

- 1. Rehashing Technique: To accommodate a colliding entry elsewhere in the hash table. Disadvantage: Clustering. Solution?
- 2. Overflow Chaining Technique: To accommodate the colliding entry in a separate table. Disadvantage: Extra memory requirement by overflow table. Solution?
- 3. Scatter Table Organization: Overcomes drawback of overflow chaining technique, i.e large memory requirement.
- Assi. Ques: Write a note on following:
  - 1. Rehashing giving suitable example (hint eg.2.6)
  - 2. Overflow Chaining with example(hint eg.2.7)
  - 3. Scatter Table Organization with example.
  - 4. Compare collision resolution techniques based on memory requirement. Also give conclusion.

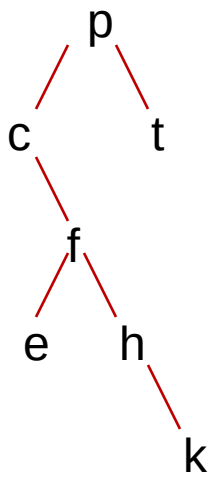
Linked List & Tree Structure Organizations

- Each entry in linked list organization contains a single pointer field.
- List has to be searched sequentially.
- Hence its performance is identical with that of sequential search tables. i.e $p_s = l/2$ and $p_u = l$.

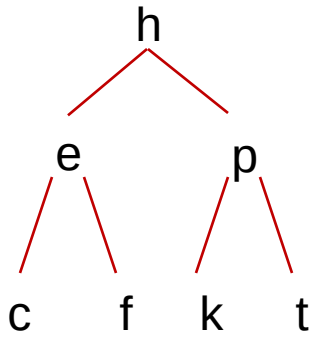
Binary Trees

- Each node in the tree is a symbol entry with two pointer fields
  i.e Left Pointer and Right Pointer.
- Algorithm 2.4 (Binary Tree Search)
- 1. current_node_pointer := address of root
- 2. if s = (current_node_pointer)*.symbol then exit with success;
- 3. if s<(current_node_pointer)*.symbol then
  current_node_pointer:= (current_node_pointer) *. left_pointer;
  else current_node_pointer:= (current_node_pointer) *. right_pointer;
- 4. if current_node_pointer := nill then
  exit with failure.
  else goto step 2.
- When can we obtain best search performance?
- Ans: when the tree is balanced.
- When the search performance is worst?
- Ans: when tree degenerates to linked list and performance becomes similar to sequential search.

Example: p,c,t,f,h,k,e.



After Rebalancing:
c, e, f, h, k, p, t



Nested Search Structures:

- Nested search structures are used when it is necessary to support a search along a secondary dimension within a search structure.
- Also called multi-list structures.
- Each symbol table entry contains two fields:
  - Field list
  - Next field

Eg: personal_info :

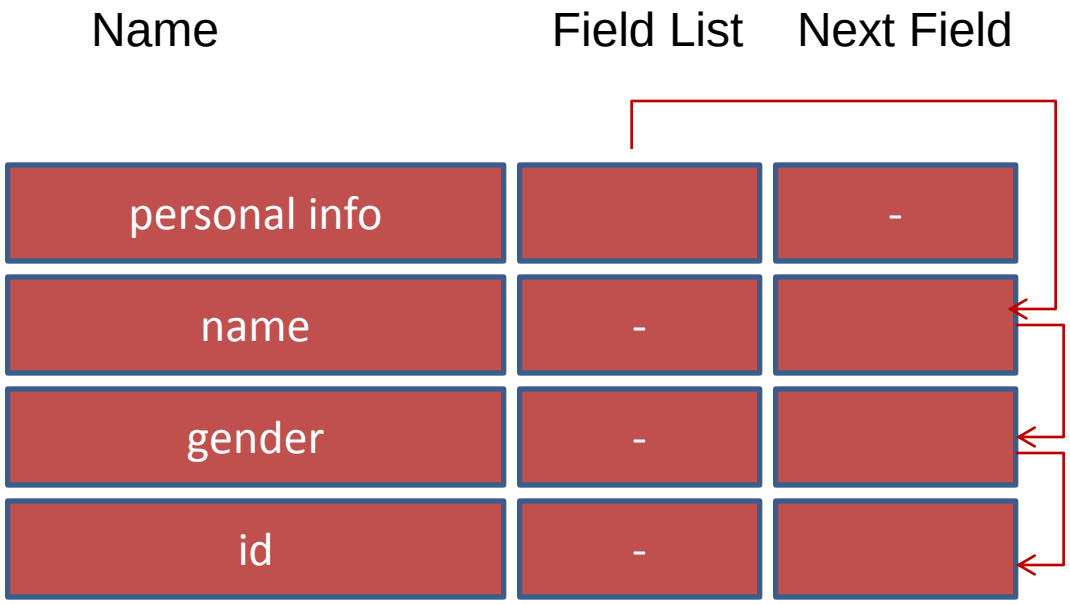
record

name : array[1..10] of char;

gender : char;

id: int;

end;

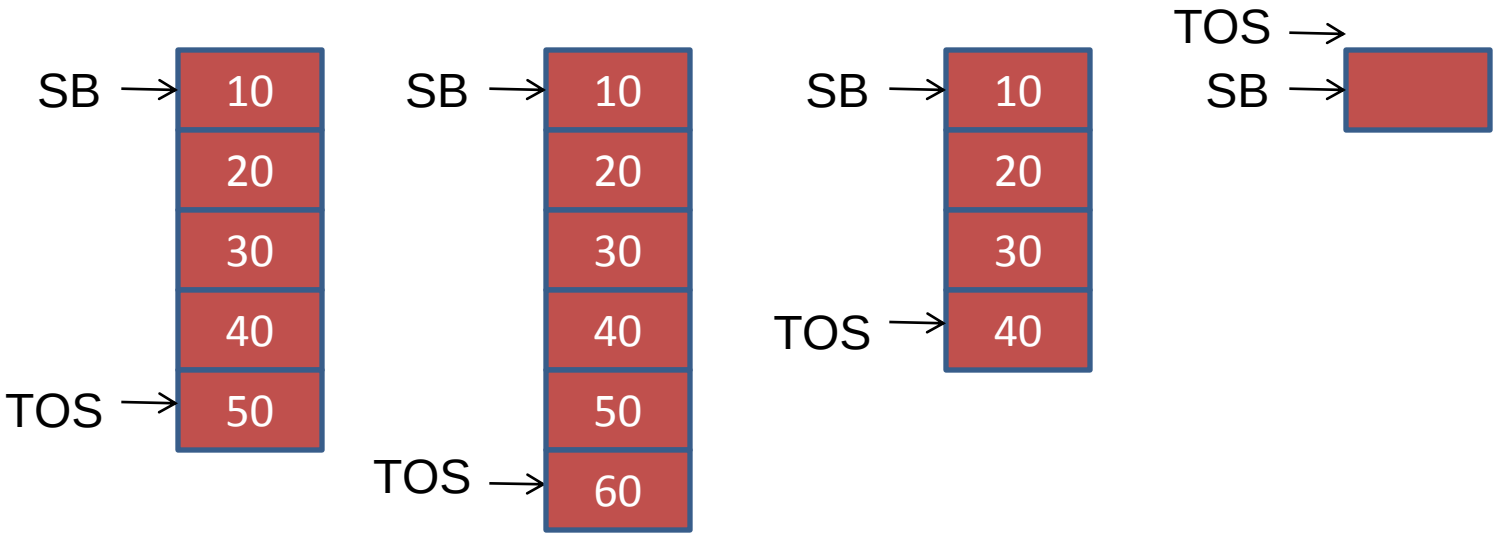


ALLOCATION DATA STRUCTURES

- Stacks
- Heaps

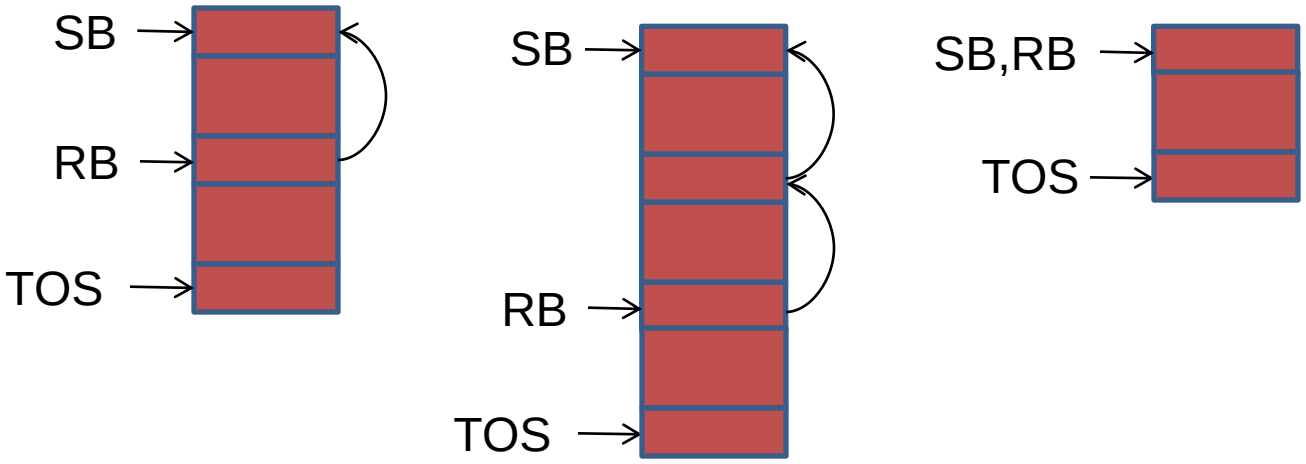
Stacks

- Is a linear data structure which satisfies following properties:
  - 1. Allocation and de-allocation are performed in a LIFO manner.
  - 2. Only last element is accessible at any time.
- SB – Stack Base points to first word of stack.
- TOS – Top Of Stack points to last entry allocated to stack.
- In last fig u can see that TOS = SB – 1.



Extended Stack Model

- All entries may not be of same size.
- Record: A set of consecutive stack entries.
- Two new pointers exist in the model other than SB and TOS.
- 1. RB Record Base pointing to the first word of the last record in stack.
- 2. ‘Reserve Pointer’, the first word of each record.
- The allocation and de-allocation time actions shown below:



Allocation

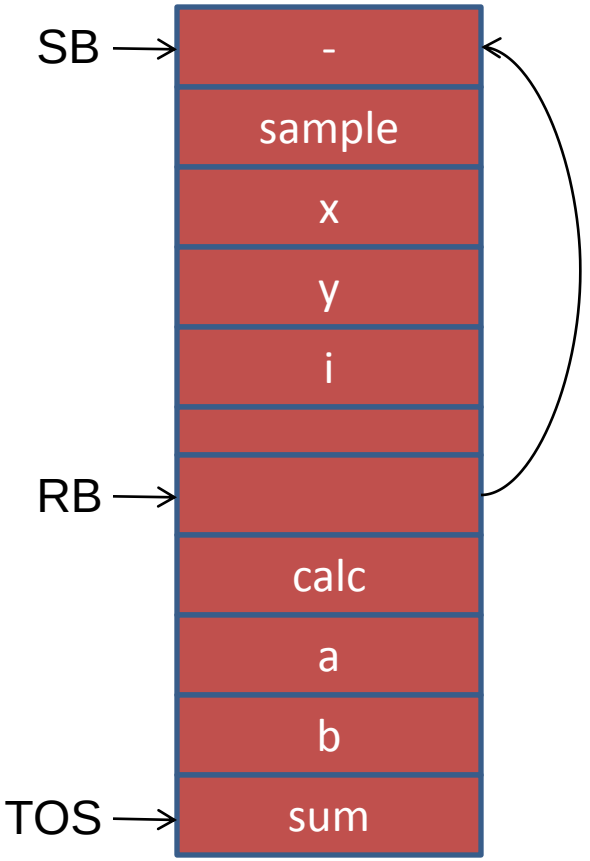
- 1. TOS := TOS + 1;
- 2. TOS* := RB;
- 3. RB := TOS;
- 4. TOS := TOS + n;
- The first statement increments TOS by one stack entry.
- Now TOS points to ‘reserved pointer’ of new record.
- 2nd statement deposits address of previous record base into ‘reserved pointer’.
- 3rd statement sets RB to point at first stack entry in the new record.
- 4th statement performs allocation of n stack entries to the new entity. See fig 2 in previous slide.
- The newly created entity now occupies the address <RB> + / to <RB> + / x n.
- RB stands for contents of Record in ‘RB’.

De-Allocation

- 1. TOS := RB – 1;
- 2. RB := RB*;
- 1st statement pops a record off the stack by resetting TOS to the value it had before the record was allocated.
- 2nd statement points RB to base of the previous record.
- That was all about allocation and de-allocation in extended stack model.
- Now let us see an implementation of this model in a Pascal program that contains nested procedures where many symbol table must co-exist during compilation.

Example: Consider Pascal Program

```
Program Sample(input,output);
   var
        x,y : real;
        i     : integer;
   Procedure calc(var a,b : real);
       var
            sum : real;
       begin
            sum := a+b;
            ---
           ---
      end calc;
   begin {Main Program}
   ----
   ----
end.
```

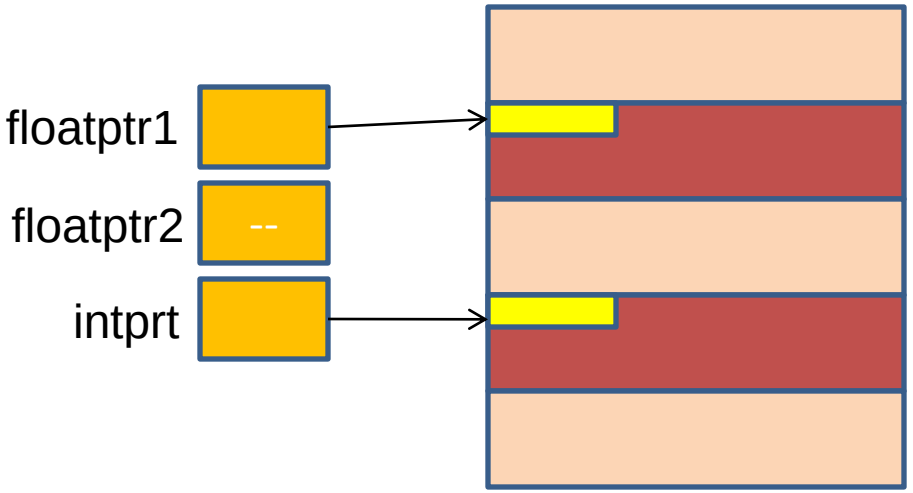


Heaps

- Non Linear Data Structure
- Permits allocation and de-allocation of entities in random order.
- Heaps DS does not provide any specific means to access an allocated entity.
- Hence, allocation request returns pointer to allocated area in heap.
- Similarly, de-allocation request must present a pointer to area to be de-allocated.
- So, it is assumed that each user of an allocated entity maintains a pointer to the memory area allocated to the entity.
- Lets take the example to clarify more what we talked.

Example: Status of Heap after c program execution

```
float *floatptr1,*floatptr2;
Int *intprt;
floatptr1=(float *)calloc(5,sizeof(float));
floatptr2=(float *)calloc(5,sizeof(float));
intptr=(int *)calloc(5,sizeof(int));
free(floatptr2);
```



- This creates a ‘hole’ in the allocation.

Memory Management

- We have seen how ‘holes’ are developed in memory due to allocation and de-allocation in the heap.
- This creates requirement of memory management that identifies free memory areas and reusing them while making fresh allocation.
- Performance criteria for memory management would be
  - Speed of allocation / de-allocation
  - Efficiency of memory utilization

Identifying Free Memory Areas

- Two popular techniques used to identify free memory areas are:
  - 1. Reference Counts
  - 2. Garbage Collection

Reference Counts

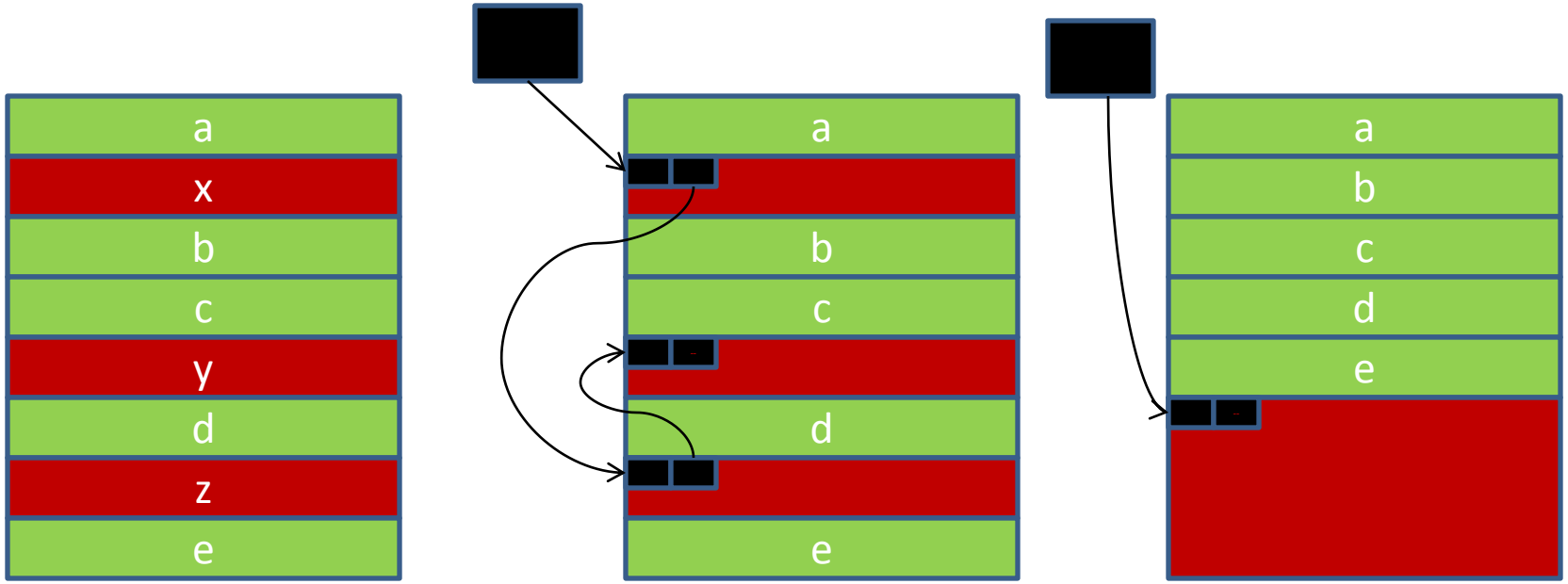
- In this technique system associates a “reference count” with each memory area to indicate the number of its active users.
- The number is incremented when new user gains access to that area.
- And the number is decremented when user finishes using it.
- The area is knows to be free when its “reference count” drops to zero.
- Advantage: Simple to implement.
- Disadvantage: Incurs Overheads at every allocation and de-allocation.

Garbage Collection

- Garbage collection makes two passes over memory to identify unused areas.
- 1st Pass: It traverses all pointers pointing to allocated areas and marks the memory areas which are in use.
- 2nd Pass: Finds all unmarked areas and declare them to be free.
- Advantage: Doesn't incur incremental overhead.
- Disadvantage: Incurred only when system runs out of free memory to allocate to fresh request, resulting to delayed performance.

Memory Compaction:

- To manage the reuse of free memory, perform memory compaction to combine these ‘free list’ areas into single ‘free area’.
- Green box indicates allocated area and Maroon box indicates de-allocated area which later gets converted to n free lists in second fig and at last compacting memory to single free list.
- First word of this area contains a count of words in area and second is next pointer which may be NULL.



Reuse of Memory:

- After memory compaction, fresh allocation can be made on free block of memory.
- Free area descriptor and count of words in free area are updated.
- When a free list is used, two techniques can be used to perform a fresh allocation:
  - 1. First Fit Technique
  - 2. Best Fit Technique

Techniques to make fresh allocation:

First Fit

- First fit technique selects first free area whose size is greater than or equal to n(number of words to be alloacated) words.
- Problem: Memory area becomes successively smaller.
- Result: Request for large memory area may have to be rejected.

Best Fit

- Best fit technique finds the smallest free area whose size is greater than or equal to n.
- Advantage: This enables more allocation request to be satisfied.
- Problem: In long run, it too may suffer from problem of numerous small free areas.

Chapter Ends Here

- Assignment Question: