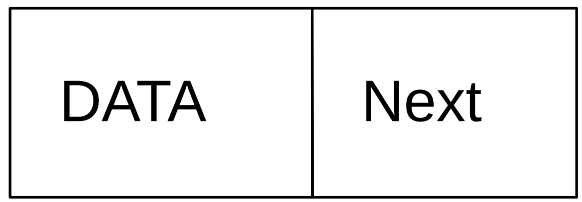
Further Data Structures

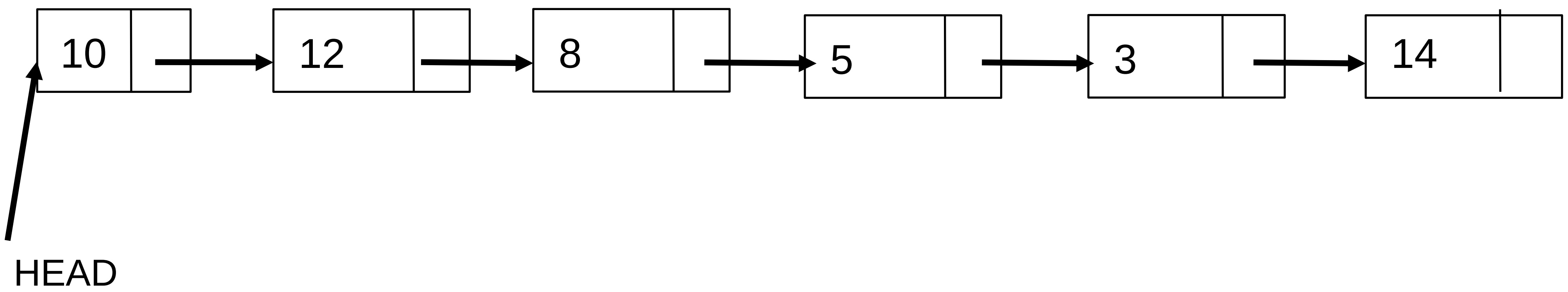
- The story so far
  - We understand the notion of an abstract data type.
  - Saw some fundamental operations as well as advanced operations on arrays.
  - Saw how restricted/modified access patterns on even arrays have several applications.
- This week we will
  - study a data structure that can grow dynamically
  - its applications

The Linked List



- The linked list is a pointer based data structure.
- Each node in the list has some data and then also indicates via a pointer the location of the next node.
  - Some languages call the pointer also as a reference.
- The node structure is as shown in the figure.

The Linked List



- How to access a linked list?
  - Via a pointer to the first node, normally called the head.
- The figure above shows an example of representing a linked list.

Basic Operations

- Think of the array. We need to be able to:
  - Add a new element
  - Remove an element
  - Print the contents
  - Find an element
- Similarly, these are the basic operations on a linked list too.

Basic Operations

- To insert, where do we insert?
- Several options possible
  - insert at the beginning of the list
  - insert at the end
  - insert before/after a given element.
- Each applicable in some setting(s).

Application – Polynomials

- Another application of linked lists is to polynomials.
- A polynomial is a sum of terms.
- Each term consists of a coefficient and a (common) variable raised to an exponent.
- We consider only integer exponents, for now.
- Example: $4x^3 + 5x - 10$.

Application – Polynomials

- How to represent a polynomial?
- Issues in representation
  - should not waste space
  - should be easy to use it for operating on polynomials.

Application – Polynomials

- Any case, we need to store the coefficient and the exponent.
- Option 1 – Use an array.
  - Index k stores the coefficient of the term with exponent k.
- Advantages and disadvantages
  - Exponent stored implicity (+)
  - May waste lot of space. When several coefficients are zero (– –)
  - Exponents appear in sorted order (+)

Application – Polynomials

- Further points
  - Even if the input polynomials are not sparse, the result of applying an operation to two polynomials could be a sparse polynomial. (--)

Application – Polynomials

```
struct node
{
    float coefficient;
    int exponent;
    struct node *next;
}
```

- Can we use a linked list?
- Each node of the linked list stores the coefficient and the exponent.
- Should also store in the sorted order of exponents.
- The node structure is as follows:

Application -- Polynomials

- How can a linked list help?
  - Can only store terms with non-zero coefficients.
  - Does not waste space.
  - Need not know the terms in a result polynomial apriori. Can build as we go.

Operations on Polynomials

- Let us now see how two polynomials can be added.
- Let P1 and P2 be two polynomials.
  - stored as linked lists
  - in sorted (decreasing) order of exponents
- The addition operation is defined as follows
  - Add terms of like-exponents.

Operations on Polynomials

- We have P1 and P2 arranged in a linked list in decreasing order of exponents.
- We can scan these and add like terms.
  - Need to store the resulting term only if it has non-zero coefficient.
- The number of terms in the result polynomial P1+P2 need not be known in advance.
- We'll use as much space as there are terms in P1+P2.

Further Operations

- Let us consider multiplication
- Can be done as repeated addition.
- So, multiply P1 with each term of P2.
- Add the resulting polynomials.

Other Applications of Linked Lists

- Sparse matrices
  - Just like sparse polynomials, sparse matrices of size nxn contain very few non-zeros.
  - How to add and multiply sparse matrices while not using an nxn matrix.
  - Use linked lists.
- Graphs: Will see later.
- Can also implement stacks and queues using linked lists.
  - Solves the problem of stack out of memory.

Further Data Structures

- The story so far
  - We understand the notion of an abstract data type.
  - Saw some fundamental operations as well as advanced operations on arrays, stacks, and queues
  - Saw a dynamic data structure, the linked list, and its applications.
- This week we will
  - focus on improving the performance of the find operation
  - Propose data structures for an efficient find.

Motivation

- Consider a high-level rogramming language such as C/C++

```
int func(int a, int b)
{
    int y = 0;
    a = a+b
     y = a/b;
     {
         int y = 1;
         a = y*b;
      }
      return y;
}
```

What is the Output?

Motivation

- Consider a high-level rogramming language such as C/C++.
- They need a compiler to translate the program.
- In that process, there are several steps and several checks.
  - One of them is to check for variable names, types, etc.
  - Ensure also that no duplicate names appear within the same scope.

```
int func(int a, int b)
{
    int y = 0;
    a = a+b
     y = a/b;
     {
         int y = 1;
         a = y*b;
      }
      return y;
}
```

What is the Output?

Motivation

- Let us consider this duplicate variable names problem.
- As we encounter a new variable declaration,
  - verify that in the same scope there are no other declarations with the same name.
  - If this is not a duplicate, need to store this name to check future declarations.
  - Once a scope is complete, can delete names from this scope.

```
int func(int a, int b)
{
    int y = 0;
    int x = 1;
    int y = 2;
    a = a+b
    y = a/b;
return y;
}
```

Motivation

test	flag	sum	● ● ●

- Let us consider a few alternatives first.
- Start with using an array.
- Store names in an array, as they appear.

Using an Array

- To insert a new variable name
  - add it to the end of the array
- To check if the new is a duplicate
  - search in the array
  - called linear search
  - Too costly at O(n) when there are n names presently.

Using an Array

- To insert a new variable name
  - add it to the end of the array
- To check if the new is a duplicate
  - search in the array
  - called linear search
  - Too costly at O(n) when there are n names presently.
- Can we keep the array sorted by variable name
  - Then can use binary search to check for a name
  - But, insertion becomes difficult.
  - Why?

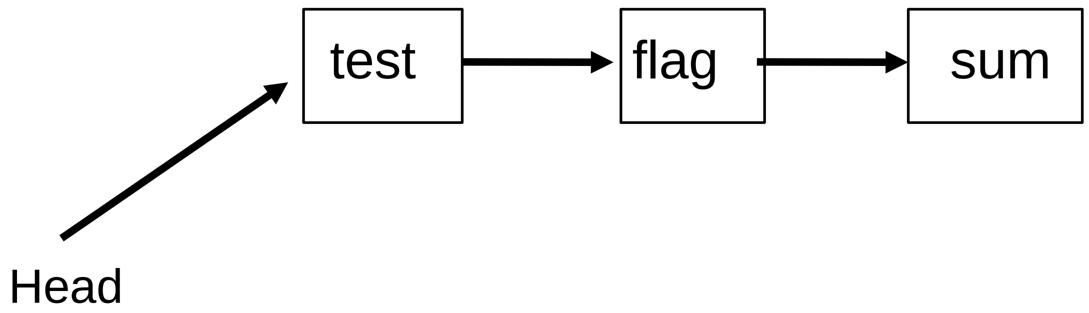
Using an Array

- So the time for ((insert, search) is:
  - (O(1), O(n)) when no sorted order
  - (O(n), O(log n)) when in sorted order.

Using a Linked List

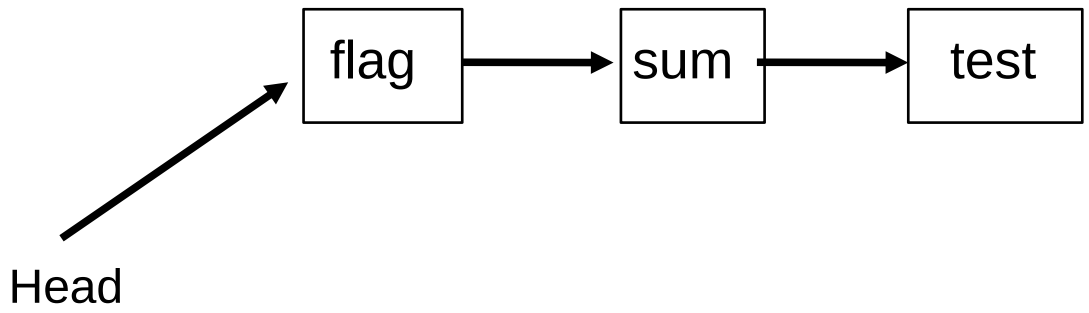
- A linked list removes the drawback that the size cannot grow dynamically.
- How would we use a linked list?

Using a Linked List



- Option 1
  - Insert names at the beginning of the list.
  - search would need to scan the entire list.
  - Time for these operations is (O(1), O(n))

Using a Linked List

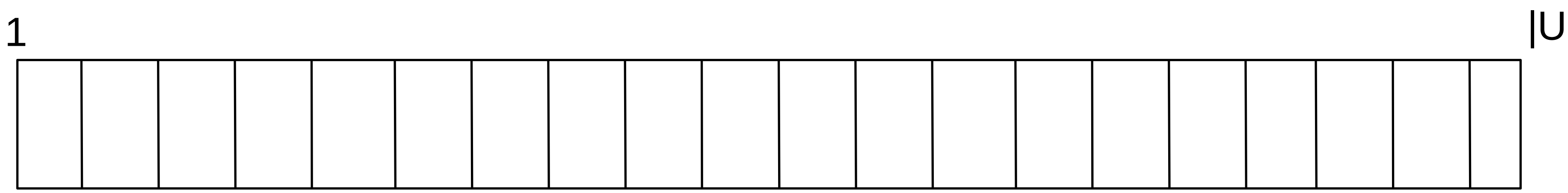


- Option 2
  - Insert names in sorted order
  - still, search would need to scan the entire list.
  - Time for these operations is (O(n), O(n))

Another Solution

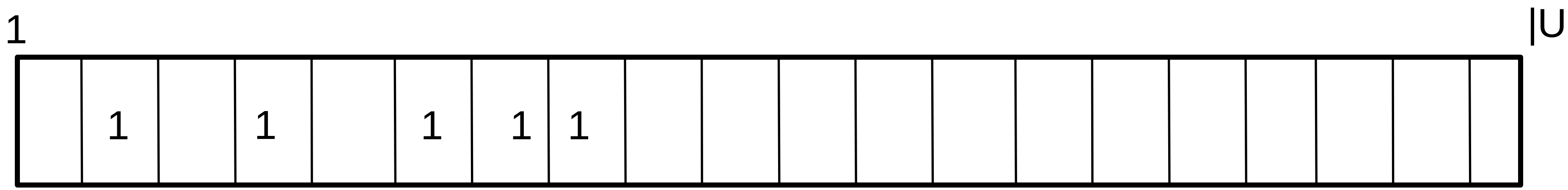
- A radically different solution for now.
- Imagine that we consider integers as the names for now.
- Let us formalize.
  - Let U be the set of all possible values. Called the universe.
  - Let K be the set of keys, a subset of U that is being used currently.

A Different Solution



- Imagine an array A of size |U|.
- Array A will have only 0 and 1 values.
- Insert an element k can be translated to setting A[k] to 1.
- Checking if k is already present would be to see if A[k] is 1 already.

A Different Solution



- Example: The following operations starting from an empty array have the effect as shown in the Figure.
  - insert(4), insert(8), insert(7), insert(2), insert(4), insert(6)
  - Empty cells assumed to contain a 0.
- Time for operations insert and search is (O(1), O(1))

A Different Solution

- has very good operation efficiency (++)
- But, can be very wasteful on space (---)
  - Imagine using such a solution for our original problem.
  - Number of valid variable names > 26^8. Why?
  - Number of variables in a typical program is about 100.
  - So, we use only 100 cells of the array of size > 26^8.
- Are there solutions so that insert, search time are both small?

A New Data Structure

- The drawback of the previous solution is that a lot of space is reserved a-priori irrespective of usage.
- Our new solution will use a space only proportional to the usage.
- Still will be based on arrays.
- Called a hash table. Details follow.

Hash Table

- Consider an array T of size |T|.
  - T is called a hash table
- Consider a function h that maps elements in U to the set {0, 1, ..., |T|-1}.
  - h is called the hash function.
- Can use the function h to map elements to indices.
  - Details follow.

Hash Table

- Now U can be any set, not just integers.
- The function h can map its input to an integer in the appropriate range.
- As an example, h(“test”) = 12.
- We will still however use integers for our setting.

Example of a Hash Table

	0									9
T :			22		34	65	76	97		

- Let U = {1, 2, ..., 100}.
- Let K = {34, 65, 22, 76, 97}.
- Let h(k) = k mod 10. So, |T| = 10.
- Key 22 to be stored in cell h(22) = 22 mod 10 = 2.
- Key 76 to be stored in cell h(76) = 76 mod 10 = 6.

Implementation of Operations

- Let us consider implementing operations insert, delete, and find.

```
operation insert(k)
begin
j = h(k)
T(j) = k;
end;
```

```
operation find(k)
begin
j = h(k)
if T(j) == k then
     return found
else return not found
end;
```

```
operation delete(k)
begin
j = h(k)
T(j) = -1;
end;
```

Operations

- Let us consider the runtime of these operations.
- All operations run in O(1) time.
  - Provided, certain conditions hold.
  - What are these conditions?
- Note the similarity to the array based solution (Solution 3)
  - Instead of accessing cell k, we now access cell h(k).
  - But, instead of using a space of |U|, we use a space of |T|.

A Small? Big Problem

	0									9
T :		91	82			25	76			

- Suppose U = {1, 2, ..., 100} as earlier.
- Suppose h(k) = k mod 10, as earlier, with |T| = 10.
- Suppose K = { 25, 76, 82, 91, 65}.
- The figure above shows the contents of T after inserting 91.
- Where should 65 be inserted?

A Small? Problem

- Notice that 65 is different from 25. So should store both.
- But, each cell of the array T can store only one element of U.
- How do we resolve this?

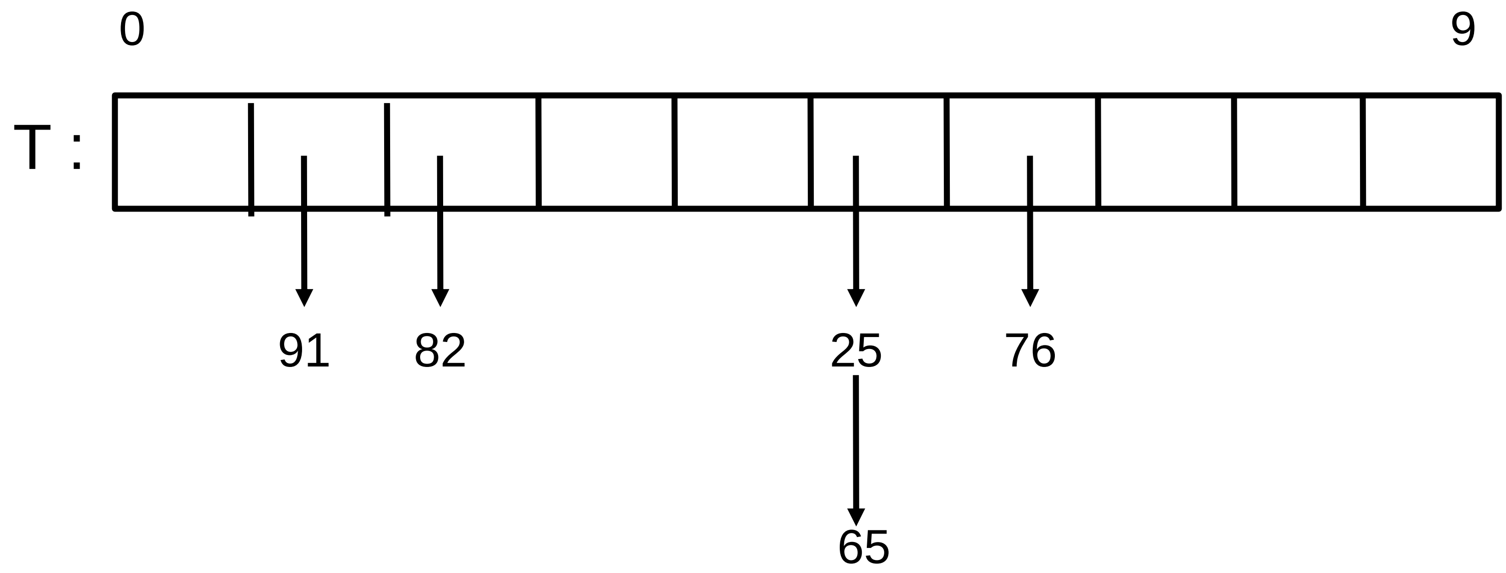
A Collision

- The situation is termed as a collision.
- Can it be avoided?
  - Not completely.
  - Notice that h maps elements of U to a range of size |T|.
  - If |U| > |T|, cannot always avoid collisions.
- Can they be minimized?
  - Certainly.
  - Choosing h() carefully can minimize collisions.
  - Some guidelines to choose h() are known.

Collision Resolution

- Despite careful efforts, it is very likely that collisions exist.
- We should have a way to handle them properly.
- Such techniques are called collision resolution techniques.
- We shall study some of those techniques.

Collision Resolution Techniques



- Can treat each cell of the table T as a pointer to a list.
- The list at cell k contains all those elements that have a hash value of k.
- Example above.

Collision Resolution Technique

- Notice how 25 and 65 are placed at the same index, 5.
- Why should 65 come after 25?
  - No reason. Several options possible.
  - Keep at the beginning of the linked list.
  - Keep at the end of the linked list.
  - Keep the linked list in sorted order.
  - Just like insertion in linked list, each has its own applications.

Collision Resolution

- The above technique is called chaining.
- Names comes from the fact that elements with the same hash value are chained together in a linked list.
- Let us see how operations should now be implemented.
  - Assuming that insert is at the front of the list.

Operations in Chaining

```
Operation insert(k)
begin
    j = T(k);
    temp = new  node;
    temp->data = k
    if T[j] == NULL then
        temp->next = NULL;
        T[j] = temp;
    else
        temp ->next = T[j];
        T[j] = temp;
end
```

```
Operation Delete(k)
begin
    j = T(k);
    while T[j] != NULL
        temp = T[j];
        if temp->data != k
            prev = temp;
            temp = temp->next;
    end-while
        prev->next = temp->next;
end
```

Operations in Chaining

- Operation Find can be similarly implemented.

Analysis of the Operations

- How to analyze the advantage of the solution?
- Consider a hash table using chaining to resolve collisions.
- What is the runtime of insert and search?