Modul Praktikum

SDAA

3

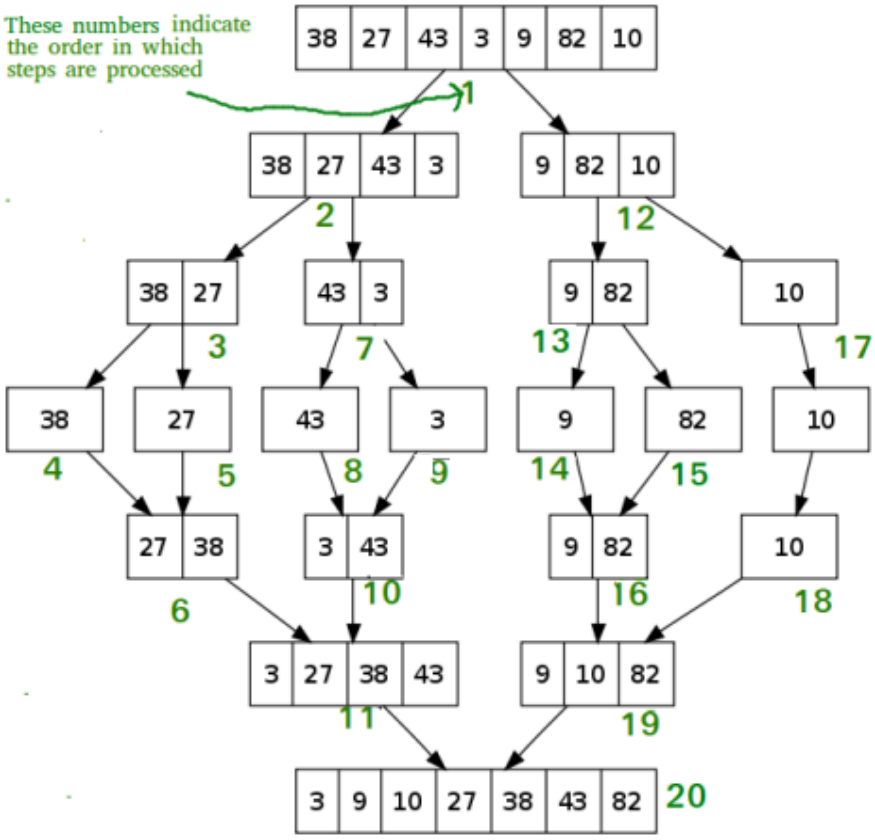
MODUL 3 PENGURUTAN / SORTING

Pengurutan data dalam struktur data sangat penting terutama untuk data yang bertipe data numerik ataupun karakter. Pengurutan dapat dilakukan secara ascending (dari yang terkecil ke terbesar) dan descending (dari yang terbesar ke terkecil). Pengurutan (Sorting) adalah proses pengurutan data yang sebelumnya disusun secara acak sehingga tersusun secara teratur menurut aturan tertentu. Dalam pengurutan data terdapat berbagai macam metode, beberapa diantaranya yaitu :

1. **Merge Sort**

   Merupakan salah satu teknik *sorting* yang menggunakan algoritma *Divide and Conquer* yang dimana cara kerja dari teknik yaitu memecah kumpulan array sehingga semua terpisah yang selanjutnya akan diurutkan dan digabungkan kembali.

   Ilustrasi Merge Sort



Source Code MergeSort

```
// C++ code for linked list merged sort
#include <iostream>
using namespace std;

/* Link list node */
struct Node {
  int data;
  Node *next;
};

/* function prototypes */
Node *SortedMerge(Node *a, Node *b);
void FrontBackSplit(Node *source, Node **frontRef, Node **backRef);
/* sorts the linked list by changing next pointers (not data) */
void MergeSort(Node **headRef){
  Node *head = *headRef;
  Node *a;
  Node *b;
  /* Base case -- length 0 or 1 */
  if ((head == NULL) || (head->next == NULL)){
    return;
  }
  /* Split head into 'a' and 'b' sublists */
  FrontBackSplit(head, &a, &b);
  /* Recursively sort the sublists */
  MergeSort(&a);
  MergeSort(&b);
  /* answer = merge the two sorted lists together */
  *headRef = SortedMerge(a, b);
}

/* See https:// www.geeksforgeeks.org/?p=3622 for details of this
function */
Node *SortedMerge(Node *a, Node *b){
  Node *result = NULL;
  /* Base cases */
  if (a == NULL)
    return (b);
  else if (b == NULL)
    return (a);
  /* Pick either a or b, and recur */
  if (a->data <= b->data){
```

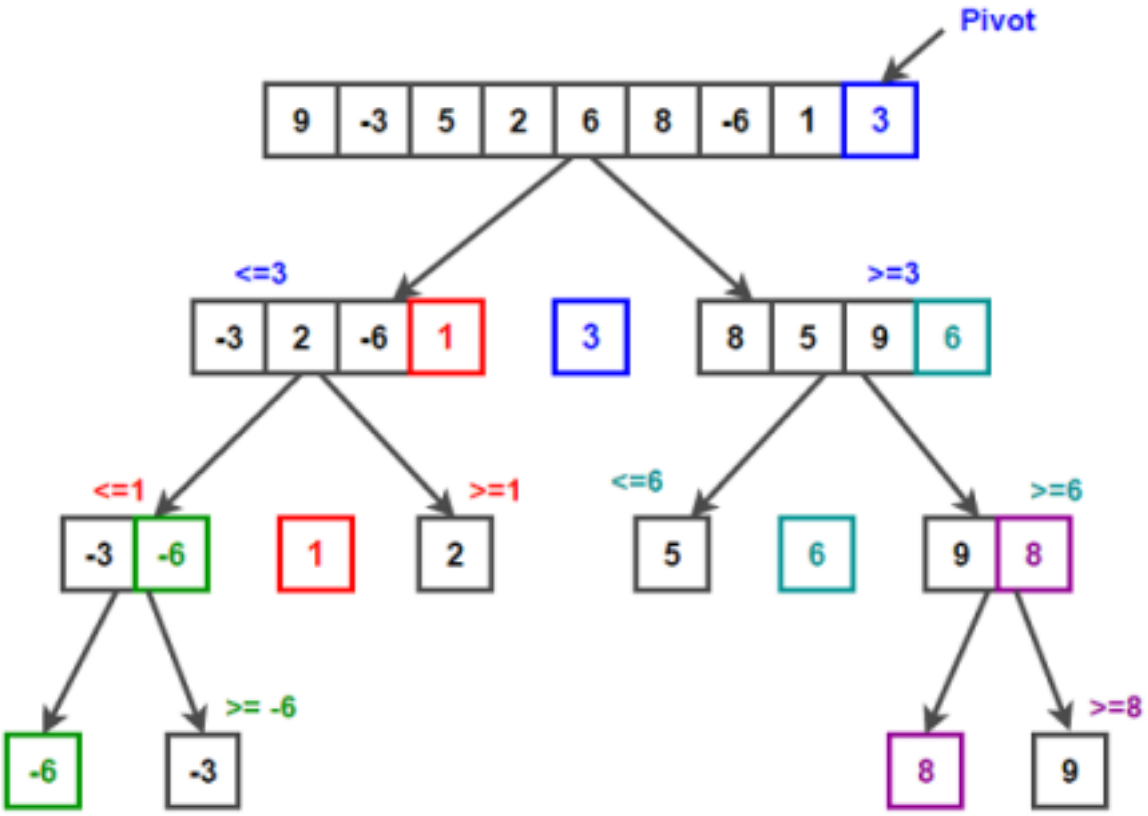
```
    result = a;
    result->next = SortedMerge(a->next, b);
  }
  else{
    result = b;
    result->next = SortedMerge(a, b->next);
  }
  return (result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
  and return the two lists using the reference parameters.
  If the length is odd, the extra node should go in the front list.
  Uses the fast/slow pointer strategy. */
void FrontBackSplit(Node *source,Node **frontRef, Node **backRef){
  Node *fast;
  Node *slow;
  slow = source;
  fast = source->next;
  /* Advance 'fast' two nodes, and advance 'slow' one node */
  while (fast != NULL){
    fast = fast->next;
    if (fast != NULL){
      slow = slow->next;
      fast = fast->next;
    }
  }
  /* 'slow' is before the midpoint in the list, so split it in two
  at that point. */
  *frontRef = source;
  *backRef = slow->next;
  slow->next = NULL;
}

/* Function to print nodes in a given linked list */
void printList(Node *node){
  while (node != NULL){
    cout << node->data << " ";
    node = node->next;
  }
}
```

```
/* Function to insert a node at the beginning of the linked list */
void insertLast(Node **head_ref, int new_data){
  /* allocate node */
  Node *new_node = new Node();
  /* put in the data */
  new_node->data = new_data;
  /* link the old list off the new node */
  new_node->next = (*head_ref);
  /* move the head to point to the new node */
  (*head_ref) = new_node;
}

/* Driver program to test above functions*/
int main(){
  /* Start with the empty list */
  Node *res = NULL;
  Node *a = NULL;
  /* Let us create a unsorted linked lists to test the functions
Created lists shall be a: 2->3->20->5->10->15 */
  insertLast(&a, 15);
  insertLast(&a, 10);
  insertLast(&a, 5);
  insertLast(&a, 20);
  insertLast(&a, 3);
  insertLast(&a, 2);
  /* Sort the above created Linked List */
  printList(a);
  MergeSort(&a);
  cout << "Sorted Linked List is: \n";
  printList(a);
  return 0;
}
// This is code is contributed by rathbhupendra
```

2. Quick Sort

Sama seperti MergeSort dimana teknik ini juga menggunakan algoritma divide and conquer dimana cara kerja dari teknik yaitu dengan mengambil salah satu elemen yang akan disebut sebagai pivot dan mempartisi (membagi) array yang berada di sekitar pivot, lalu akan dilakukan pengurutan berdasarkan perbandingan elemen yang lebih kecil atau lebih besar dari pivot.

Ilustrasi Quick Sort



Source Code QuickSort

```
// C++ program for Quick Sort on Singly Linked List

#include <cstdio>
#include <iostream>
using namespace std;

/* a node of the singly linked list */
struct Node{
  int data;
  struct Node *next;
};

/* A utility function to insert a node at the beginning of
 * linked list */
void push(struct Node **head_ref, int new_data){
  /* allocate node */
  struct Node *new_node = new Node;
  /* put in the data */
  new_node->data = new_data;
```

```
  /* link the old list off the new node */
  new_node->next = (*head_ref);
  /* move the head to point to the new node */
  (*head_ref) = new_node;
}

/* A utility function to print linked list */
void printList(struct Node *node){
  while (node != NULL){
  printf("%d ", node->data);
  node = node->next;
  }
  printf("\n");
}

// Returns the last node of the list
struct Node *getTail(struct Node *cur){
  while (cur != NULL && cur->next != NULL){
    cur = cur->next;
  }
  return cur;
}

// Partitions the list taking the last element as the pivot
struct Node *partition(struct Node *head, struct Node *end, struct
Node **newHead, struct Node **newEnd){
  struct Node *pivot = end;
  struct Node *prev = NULL, *cur = head, *tail = pivot;
  // During partition, both the head and end of the list
  // might change which is updated in the newHead and
  // newEnd variables
  while (cur != pivot){
    if (cur->data < pivot->data){
      // First node that has a value less than the
      // pivot - becomes the new head
      if ((*newHead) == NULL){
        (*newHead) = cur;
      }
      prev = cur;
      cur = cur->next;
    } else { // If cur node is greater than pivot
      // Move cur node to next of tail, and change
      // tail
```

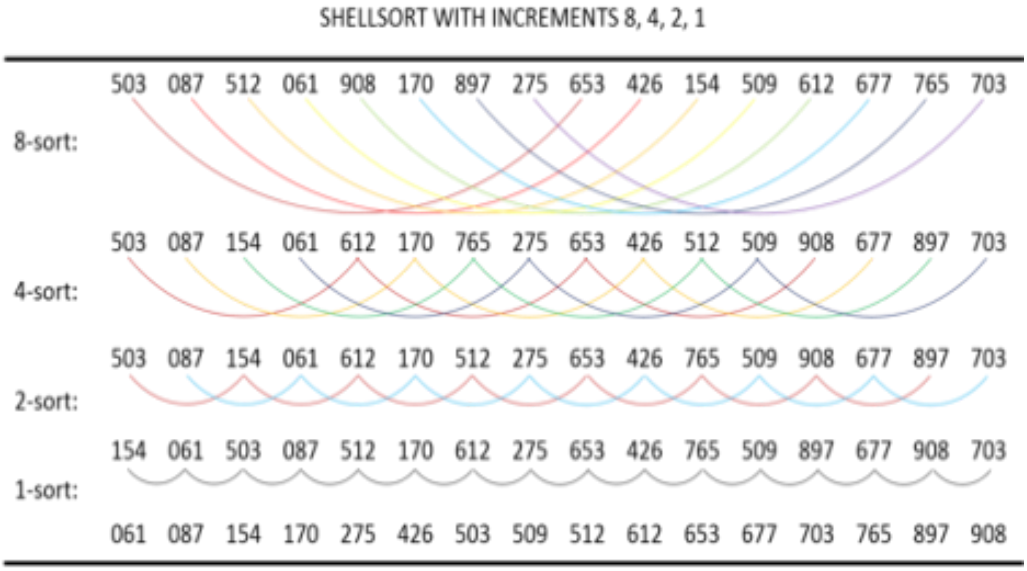
```
      if (prev){
        prev->next = cur->next;
      }
      struct Node *tmp = cur->next;
      cur->next = NULL;
      tail->next = cur;
      tail = cur;
      cur = tmp;
    }
  }
  // If the pivot data is the smallest element in the
  // current list, pivot becomes the head
  if ((*newHead) == NULL){
    (*newHead) = pivot;
  }
  // Update newEnd to the current last node
  (*newEnd) = tail;
  // Return the pivot node
  return pivot;
}

// here the sorting happens exclusive of the end node
struct Node *quickSortRecur(struct Node *head, struct Node *end){
  // base condition
  if (!head || head == end){
    return head;
  }
  Node *newHead = NULL, *newEnd = NULL;
  // Partition the list, newHead and newEnd will be
  // updated by the partition function
  struct Node *pivot = partition(head, end, &newHead, &newEnd);
  // If pivot is the smallest element - no need to recur
  // for the left part.
  if (newHead != pivot){
    // Set the node before the pivot node as NULL
    struct Node *tmp = newHead;
    while (tmp->next != pivot){
      tmp = tmp->next;
      tmp->next = NULL;
      // Recur for the list before pivot
      newHead = quickSortRecur(newHead, tmp);
      // Change next of last node of the left half to
      // pivot
```

```
      tmp = getTail(newHead);
      tmp->next = pivot;
    }
    // Recur for the list after the pivot element
    pivot->next = quickSortRecur(pivot->next, newEnd);
    return newHead;
}

// The main function for quick sort. This is a wrapper over
// recursive function quickSortRecur()
void quickSort(struct Node **headRef){
  (*headRef) = quickSortRecur(*headRef, getTail(*headRef));
  return;
}

// Driver's code
int main(){
  struct Node *a = NULL;
  push(&a, 5);
  push(&a, 20);
  push(&a, 4);
  push(&a, 3);
  push(&a, 30);
  cout << "Linked List before sorting \n";
  printList(a);
  // Function call
  quickSort(&a);
  cout << "Linked List after sorting \n";
  printList(a);
  return 0;
}
```

3. **Shell Sort**

Merupakan algoritma sorting varian dari insertion sort yang dimana perbedaannya dengan insertion sort yaitu jika di insertion sort hanya dapat menukar elemen satu langkah kedepan sedangkan di shell sort penukaran elemen dapat dilakukan dengan jarak yang lebih jauh.

Ilustrasi Shell Sort



Source Code ShellSort

```
// C++ implementation of Shell Sort
#include <iostream>
using namespace std;

struct Node{
     int data;
     Node *next;
};
/* function to sort arr using shellSort */
int length(Node *head){
     int panjang = 0;
     Node *temp = head;
     while (temp != NULL){
       temp = temp->next;
       panjang++;
     }
     return panjang;
}
int linkedList2Array(Node *head,int *arr){
     // Cek banyak data
     int panjang = length(head);
     for (int i = 0; i < panjang; i++){
       arr[i] = head->data;
       head = head->next;
     }
```

```
    return panjang;
}
int array2LinkedList(Node *head,int *arr){
    // Cek banyak data
    int panjang = length(head);
    for (int i = 0; i < panjang; i++){
      head->data = arr[i];
      head = head->next;
    }
    return panjang;
}
int shellSort(int arr[], int n){
    // Start with a big gap, then reduce the gap
    for (int gap = n / 2; gap > 0; gap /= 2){
      // Do a gapped insertion sort for this gap size.
      // The first gap elements a[0..gap-1] are already in gapped order
      // keep adding one more element until the entire array is
      // gap sorted
      for (int i = gap; i < n; i += 1){
            // add a[i] to the elements that have been gap sorted
            // save a[i] in temp and make a hole at position i
            int temp = arr[i];

            // shift earlier gap-sorted elements up until the correct
            // location for a[i] is found
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                  arr[j] = arr[j - gap];

            // put temp (the original a[i]) in its correct location
            arr[j] = temp;
      }
    }
    return 0;
}

void display(Node *head){
    while(head != NULL){
      cout << head->data << " ";
      head = head->next;
    }
    cout << endl;
}
```

```
int main(){
    Node *node1 = new Node;
    Node *node2 = new Node;
    Node *node3 = new Node;
    Node *node4 = new Node;
    Node *node5 = new Node;
    node1->data = 2;
    node2->data = 1;
    node3->data = 4;
    node4->data = 3;
    node5->data = 5;
    node1->next = node2;
    node2->next = node3;
    node3->next = node4;
    node4->next = node5;
    node5->next = NULL;
    Node *HEAD = node1;
    // Pindahkan data ke dalam array
    int arr[length(HEAD)];
    linkedList2Array(HEAD,arr);

    cout << "Sebelum disorting";
    display(HEAD);
    // Sorting data array
    shellSort(arr, length(HEAD));

    // Kembalikan data ke dalam linked list
    array2LinkedList(HEAD,arr);
    cout << "Setelah disorting";
    display(HEAD);
    return 0;
}
```