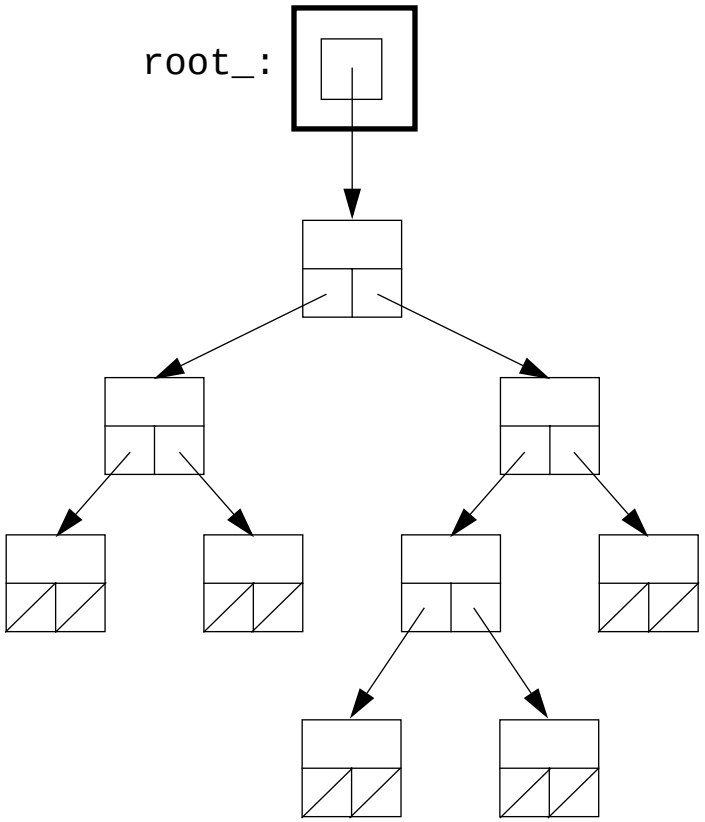
Binärt träd

```
struct Tree_Node
{
public:
   Tree_Node(const std::string& value) : data_{ value } {}

   Tree_Node(const std::string& value,
             Tree_Node* left, Tree_Node* right)
     : data_{ value }, left_{ left }, right_{ right }
   {}

   ~Tree_Node() { delete left_; delete right_; }

   std::string data_;

   Tree_Node* left_{ nullptr };
   Tree_Node* right_{ nullptr };
};


class Tree
{
public:
   …
private:
   Tree_Node* root_{ nullptr };
};
```



Trädtraversering

Besöka alla noder i trädet på ett systematiskt sätt, olika varianter:

- djupet först
  - vänster-till-höger eller höger-till-vänster
  - preorder (prefix) – inorder (infix) – postorder (postfix)
- bredden först (nivåtraversering)
  - kan göras med hjälp av en kö

Trädtraversering djupet-först

```
void traverse(Tree_Node* tree, void (*op)(Tree_Node*))
{
    if (tree)
    {
        traverse(tree->left_, op);

        op(tree);

        traverse(tree->right_, op);
    }
}
```

- vänster-till-höger
- inorder (infix) traversering
  - vänster subträd behandlas
  - noden ifråga behandlas
  - höger subträd behandlas
- operationen som ska utföras på varje nod skickas med i form av en funktionspekare

Trädtraversering bredden-först

```
void traverse(Tree_Node* tree, void (*op)(Tree_Node*))
{
    if (tree)
    {
        std::queue<Tree_Node*> nodes;
        nodes.push(tree);

        while (!nodes.empty())
        {
            Tree_Node* current_node = nodes.front();
            nodes.pop();

            if (current_node->left_ ) nodes.push(current_node->left_);
            if (current_node->right_ ) nodes.push(current_node->right_);

            op(current_node);
        }
    }
}
```

- vänster till höger
- nivå för nivå
- tomma träd ignoreras
  - ett alternativ kan vara att köa även tompekare

Bestämma djupet hos ett träd

Innebär också att alla noder kommer att besökas.

```
int depth(const Tree_Node* tree)
{
   if (tree)
      return 1 + max(tree->depth(tree->left_), tree->depth(tree->right_));
   else
      return 0;
}
```

- rotnoden i ett träd har per definition djupet 0
  - djupet för en nod är väglängden från roten till noden
  - ett löv har per definition *höjd* 0
  - höjden för rotnoden är den längsta väglängden till något av löven

Trädsökning – rekursiv lösning

Innebär att följa en specifik väg ner genom trädet.

```
Tree_Node* find(std::string x, Tree_Node* tree)
{
   if (tree && tree->data_ != x)
   {
      if (x < tree->data_)
         return find(x, tree->left_);
      else
         return find(x, tree->right_);
   }

   return tree;
}
```

- endast noder utmed en specifik sökväg besöks

Trädsökning – iterativ lösning

Enkel och utan rekursion.

```
Tree_Node* find(std::string, Tree_Node* tree)
{
   while (tree && tree->data_ != x)
   {
      if (x < tree->data_)
         tree = tree->left_;
      else
         tree = tree->right_;
   }

   return tree;
}
```

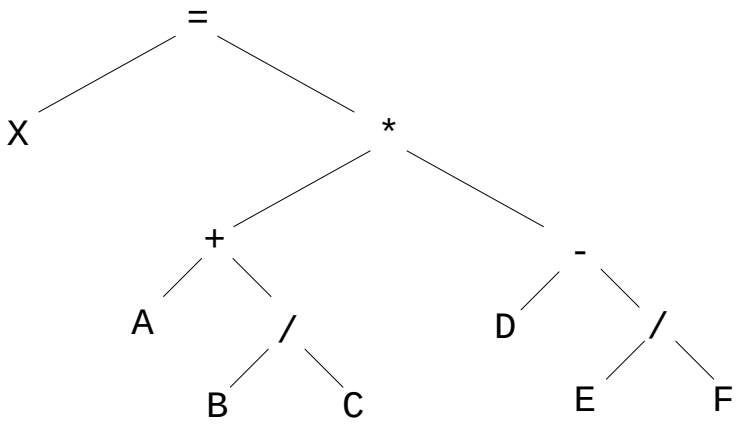
- iteration är normalt att föredra jämfört med rekursion fall som detta – lika enkel algoritm

Uttryckträd

Ett tilldelningsuttryck som

```
X = (A + B / C) * (D - E / F)
```

representeras av följande uttryckträd.



- alla operander är löv – de finns i samma ordning från vänster till höger i trädet som i uttrycket
- alla operatorer är inre noder – binära operatorer med sina två operander som vänster respektive höger subträd
- varje subträd representerar ett deluttryck – parenteser behövs inte

Vilket resultat ger respektive variant av trädtraversering enligt djupet först med följande operation?

```
void print(Tree_Node* t) { cout << t->data_ << ' '; }
```

Standardundantag

`exception`			***exempelvis kastad av/om:***
	`logic_error`		
		`domain_error`	*otillåtna funktionsvärden*
		`invalid_argument`	`bitset`-*konstruktor*
		`length_error`	*objekts längd överskrids*
		`out_of_range`	`at()`
		`future_error`	*funktioner i trådbiblioteket*
	`runtime_error`		
		`range_error`	*vissa beräkningar*
		`overflow_error`	`bitset::to_long()`
		`underflow_error`	*vissa beräkningar*
		`system_error`	*systemrelaterade funktioner*
			`ios_base::failure` `ios_base::clear()`
	`bad_typeid`		**`typeid`**
	`bad_cast`		**`dynamic_cast`**
	`bad_weak_pointer`		`std::shared_ptr`-*konstruktorer*
	`bad_exception`		*brott mot exceptionspecification*
	`bad_function_call`		`std::function::operator()`
	`bad_alloc`		**`new`**
		`bad_array_new_length`	**`new`**`[]`

Standardundantagen

- klassen `exception` är basklass för nästan alla andra standardundantag
- en del direkta subklasser till exception används för att kasta undantag
  - `bad_exception`, t.ex.
- `logic_error` representerar sådant som beror på fel i programmets logik och i teorin förebyggbart
  - `domain_error, invalid_argument, length_error, out_of_range`
- `runtime_error` representerar sådant som beror på fel utanför programmets kontroll och inte enkelt kan förutses
  - `range_error, overflow_error, underflow_error, system_error`
- några undantagsklasser är inte härledda från `exception`
  - `nested_exception`, t.ex.

Basklassen exception

```
class exception
{
public:
    exception() noexcept;
    exception(const exception& e) noexcept;

    virtual ~exception();

    exception& operator=(const exception& e) noexcept;

    virtual const char* what() const noexcept;
};
```

- polymorf klass
  - inte abstrakt men bör endast användas som basklass för mer användbara undantagsklasser
  - defaultkonstruktor
  - kopieringskonstruktor
  - virtuell destruktor (kastar per definition inte undantag)
  - kopieringstilldelningsoperator
  - virtuell funktion what()
- **noexcept** specificerar att funktionen inte kastar undantag
  - viktigt att undantag inte i sin tur kan kasta nya undantag

Regler för generering av flyttkonstruktor och flyttilldelningsoperator

Klassen exception är en bra klass för att ta upp detta.

- *flyttkonstruktor* genereras endast om klassen
  - *inte* har en användardeklarerad kopieringskonstruktor
  - *inte* har en användardeklarerad kopieringstilldelningsoperator
  - *inte* har en användardeklarerad flyttilldelningsoperator
  - *inte* har en användardeklarerad destruktor
- *flyttilldelningsoperator* genereras endast om klassen
  - *inte* har en användardeklarerad kopieringskonstruktor
  - *inte* har en användardeklarerad flyttkonstruktor
  - *inte* har en användardeklarerad kopieringstilldelningsoperator
  - *inte* har en användardeklarerad destruktor
- exception har en användardeklarerad kopieringskonstruktor, kopieringstilldelningsoperator och destruktor
  - varken flyttkonstruktor eller flyttilldelningsoperator genereras alltså

En av de direkta subklasserna – logic_error

Standarden anger endast konstruktorerna men i praktiken definieras logic_error så här.

```
class logic_error : public std::exception
{
public:
    explicit logic_error(const std::string& what_arg) : msg_(what_arg) {}

    explicit logic_error(const char* what_arg) : msg_(what_arg) {}

    virtual const char* what() const noexcept { return msg_.data(); }

private:
    string msg_;
};
```

- defaultkonstruktor genereras *inte*, eftersom en annan konstruktor har deklarerats
- kopieringstilldelningsoperator genereras
- flyttkonstruktor och flyttilldelningsoperator genereras *inte*, eftersom kopieringskonstruktorn deklareras
- destruktor kommer att genereras och vara **virtual**
- what() överskuggar what() deklarerad i exception

En av subklasserna till logic_error – length_error

```
class length_error : public std::logic_error
{
public:
   explicit length_error(const std::string& what_arg) : std::logic_error(what_arg) {}

   explicit length_error(const char* what_arg) : std::logic_error(what_arg) {}
};
```

- defaultkonstruktor genereras *inte*, eftersom en annan konstruktor har deklarerats
- kopieringstilldelningsoperator genereras
- flyttkonstruktor och flyttilldelningsoperator genereras *inte*, eftersom en kopieringskonstruktor deklareras
- destruktor genereras
- what() ärvs som den är från logic_error

Egendefinierad undantagsklass

Härled från exempelvis logic_error

```
class some_error : public std::logic_error
{
public:
   explicit some_error(const std::string& what_arg) : std::logic_error(what_arg) {}

   explicit some_error(const char* what_arg) : std::logic_error(what_arg) {}
};
```

Dessutom följande genererade eller ärvda funktionalitet:

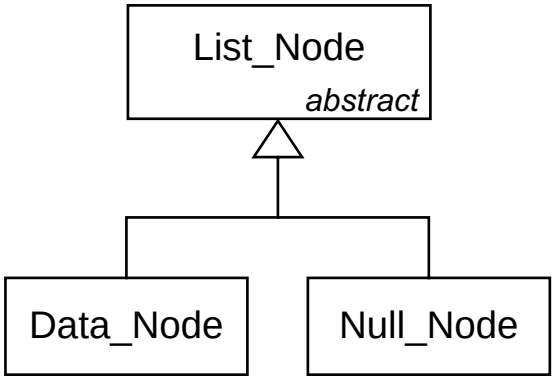
- kopieringskonstruktor
- kopieringstilldelningsoperator
- destruktor
- what()

Hantering av undantag

```
try
{
   do
   {
      cout << "Ange en radposition, avsluta med -1: ";
      cin >> pos;
      cout << line.at(pos) << endl;
   }
   while (pos > -1);
}
catch (const out_of_range& e)
{
   cout << e.what() << endl;
}
catch (const exception& e)
{
   cout << e.what() << endl;
}
catch (...)
{
   cout << "Ett oväntat fel har inträffat" << endl;
}
```

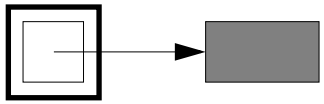
- **catch**-hanterarna går från specialiserade till mer generella – `out_of_range` – `exception` – *vad-som-helst*

Länkad lista med polymorfa listnoder

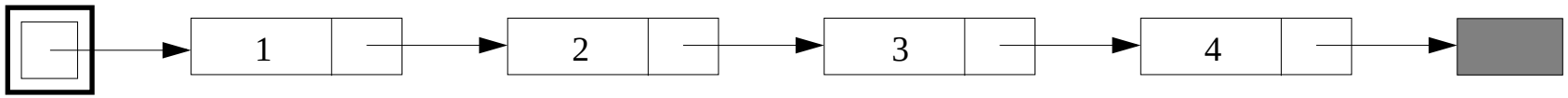


Konstruerad som en containerklass med två slags listnoder

- `Data_Node` – lagrar de data som satts i i en lista
- `Null_Node` – anger slutet av en lista, lagrar inga data
- `Data_Node` och `Null_Node` är härledda från en gemensam basklass `List_Node`
- `List` – klass som representerar en *ordnad lista* – lagrar en pekare till
  - Null_Node om listan är tom



  - Data_Node om listan innehåller ett eller flera värden



- alla speciella medlemsfunktioner för kopiering elimineras för listnodklasserna
  - endast polymorf kopiering med funktionen clone() tillåts
- operationer är implementerade med hjälp av polymorfi och rekursion
- iteratorer ska finnas för List men har utelämnats i exemplen (motsvarande uppgift ska göras i laboration Listan)

List_Node – abstrakt, polymorf basklass för listnoder

```
struct List_Node
{
   List_Node() = default;                                   //  används av Null_Node()
   List_Node(const List_Node&) = delete;

   virtual ~List_Node() = default;

   List_Node& operator=(const List_Node&) = delete;

   virtual List_Node* insert(int value) = 0;
   virtual List_Node* remove(int value) = 0;

   virtual const List_Node* find(int value) const = 0;

   virtual int  length() const = 0;
   virtual bool empty() const = 0;

   virtual List_Node* clone() const = 0;
};
```

Data_Node – nodtyp för att lagra ett insatt värde

```
struct Data_Node : public List_Node
{
   Data_Node(int value, List_Node* next) : data_{ value }, next_{ next } {}
   Data_Node(const Data_Node&) = delete;

   ~Data_Node() { delete next_; }

   Data_Node& operator=(const Data_Node&) = delete;

   Data_Node* insert(int value) override;
   List_Node* remove(int value) override;

   const List_Node* find(int value) const override;

   int  length() const override;
   bool empty() const override;

   Data_Node* clone() const override;

   int        data_;
   List_Node* next_;
};
```

Observera *– i den givna koden är medlemsfunktioner som är enkla definierade i klassen.*

Null_Node – markerar slutet på en lista

```
struct Null_Node final : public List_Node
{
   Null_Node() = default;                                    //  används av List
   Null_Node(const Null_Node&) = delete;

   ~Null_Node() = default;

   Null_Node& operator=(const Null_Node&) = delete;

   Data_Node* insert(int value) override;
   Null_Node* remove(int value) override;

   const List_Node* find(int value) const override;

   int  length() const override;
   bool empty() const override;

   Null_Node* clone() const override;
};
```

Observera *– i den givna koden är alla medlemsfunktioner definierade i klassen.*

List – en containerklass

```
class List
{
public:
   List() : list_{ new Null_Node } {}
   List(const List& other) : list_{ copy(other.list_) }{}
   List(List&& other) noexcept : List() { swap(other); }          // delegering till List()

   ~List() { delete list_; }

   List& operator=(const List& rhs) &;
   List& operator=(List&& rhs) & noexcept;

   void insert(int value);
   void remove(int value);
   bool member(int value) const;
   int  length() const;
   bool empty() const;
   void clear();

   void swap(List& other) noexcept;                               // Även icke-medlem-swap finns

   // Iteratorer…
private:
   List_Node* list_;

   static List_Node* copy(const List_Node* p);
};
```

En containerklass ska alltid ha swap-funktioner

Medlem

```
void List::swap(List& other)
{
   std::swap(list_, other.list_);
}
```

Icke-medlem

```
void swap(List& a, List& b)
{
   a.swap(b);
}
```

Argumenten till std::swap() är pekare – standarden garanterar att undantag inte kastas.

Radera en listas innehåll

```
void List::clear()
{
    if (!empty())
    {
        Data_Node* garbage = dynamic_cast<Data_Node*>(list_);     // pekare till datanoderna i listan
        Data_Node* p = garbage;

        while (!dynamic_cast<Null_Node*>(p->next_))
        {
            p = dynamic_cast<Data_Node*>(p->next_);
        }

        // p pekar på datanoden direkt före null-noden

        list_ = p->next_;                                         // behåll null-noden
        p->next_ = nullptr;                                       // länka ur den
        delete garbage;                                           // radera alla datanoder
    }
}
```

- list_ har typen List_Node*
  - måste typomvandlas till Data_Node* för att kunna initiera garbage
- datamedlemmen next_ har typen List_Node*
  - måste typomvandlas till Data_Node* för att kunna tilldelas p inuti **while**-satsen
- typomvandlingen i **while**-satsen styruttryck kontrollerar om nästa nod är null-noden

List – konstruktorer och destruktor

Defaultkonstruktor

```
List() : list_{ new Null_Node } {}
```

- initierar listan till tom lista – en Null_Node

Kopieringskonstruktor

```
List(const List& other) : list_{ copy(other.list_) }{}
```

- kopierar listan av noder i `other`

Flyttkonstruktor

```
List(List&& other) noexcept : List() { swap(other); }
```

- initierar det nya `List`-objektet till tom lista och byter sedan innehåll med `other`
- delegerar initiering av `list_` till `List()` innan bytet med `swap()`
- ska deklareras **noexcept**

Destruktor

```
~List() { delete list_; }
```

- rekursiv destruering av noderna i listan
- när detta är gjort upphör `List`-objektet att existera

List – tilldelningsoperatorer

Kopieringstilldelningsoperator

```
List& List::operator=(const List& rhs) &
{
    List{ rhs }.swap(*this);
    return *this;
}
```

- *ref-qualifier* & – tilldelning kan endast användas på vänsterargument som är *lvalue* (ska vara en variabel)
- implementeras med idiomet skapa en temporär och byt
- undantag kan kastas när det temporära objektet ska initieras – startk undantagssäkert dock
- ska deklareras med ref-qualifer &

Flyttilldelningsoperator

```
List& List::operator=(List&& rhs) & noexcept
{
    clear();
    swap(rhs);
    return *this;
}
```

- vänsterargumentets innehåll raderas – blir tom lista
- vänsterargumentet och högerargumentet byter innehåll – högerargumentet blir en tom lista
- ska deklareras med ref-qualifer & och **noexcept**

Intern kopiering av lista

```
List_Node* List::copy(const List_Node* p)
{
    return p->clone();      // clone() kan kasta undantag
}


Data_Node* Data_Node::clone() const
{
    //  Att göra – Om undantag kastas här läcker minnet för den redan kopierade svansen!
    return new Data_Node{ data_, next_->clone() };
}


Null_Node* Null_Node::clone() const
{
    //  Att göra – Om undantag kastas här läcker något minne?
    return new Null_Node;
}
```

Observera `- override` används inte utanför klassen.

Listans längd

```
int List::length() const

{
   return list_->length();
}


int Data_Node::length() const
{
   return 1 + next_->length();
}


int Null_Node::length() const
{
   return 0;
}
```

Finns ett visst värde i listan?

```
bool List::member(int value) const
{
   return list_->find(value) != nullptr;
}


const List_Node* Data_Node::find(int value) const
{
   if (data_ == value)
      return this;
   else
      return next_->find(value);
}


const List_Node* Null_Node::find(int) const              // Deklarera ej namn för parameter
{
   return nullptr;
}
```

Är listan tom?

```
bool List::empty() const
{
    return list_->empty();
}


bool Data_Node::empty() const
{
    return false;
}


bool Null_Node::empty() const
{
    return true;
}
```

Sätta in ett värde i listan

```
void List::insert(int value)
{
    list_ = list_->insert(value);
}


Data_Node* Data_Node::insert(int value)
{
    if (value < data_)
    {
        return new Data_Node{ value, this };        // Om undantag kastas läcker minne?
    }
    else
    {
        next_ = next_->insert(value);
        return this;
    }
}


Data_Node* Null_Node::insert(int value)
{
    return new Data_Node{ value, this };            // Om undantag kastas läcker minne?
}
```

Ta bort värde ur listan (det först hittade)

```
void List::remove(int value)
{
    list_ = list_->remove(value);
}

List_Node* Data_Node::remove(int value)
{
    if (value == data_)
    {
        List_Node* next{ next_ };
        next_ = nullptr;                    //  Rekursiv destruktor – noden måste länkas ur helt innan destruering
        delete this;
        return next;
    }
    else
    {
        next_ = next_->remove(value);
        return this;
    }
}


Null_Node* Null_Node::remove(int)           //  Värdet fanns inte
{
    return this;
}
```