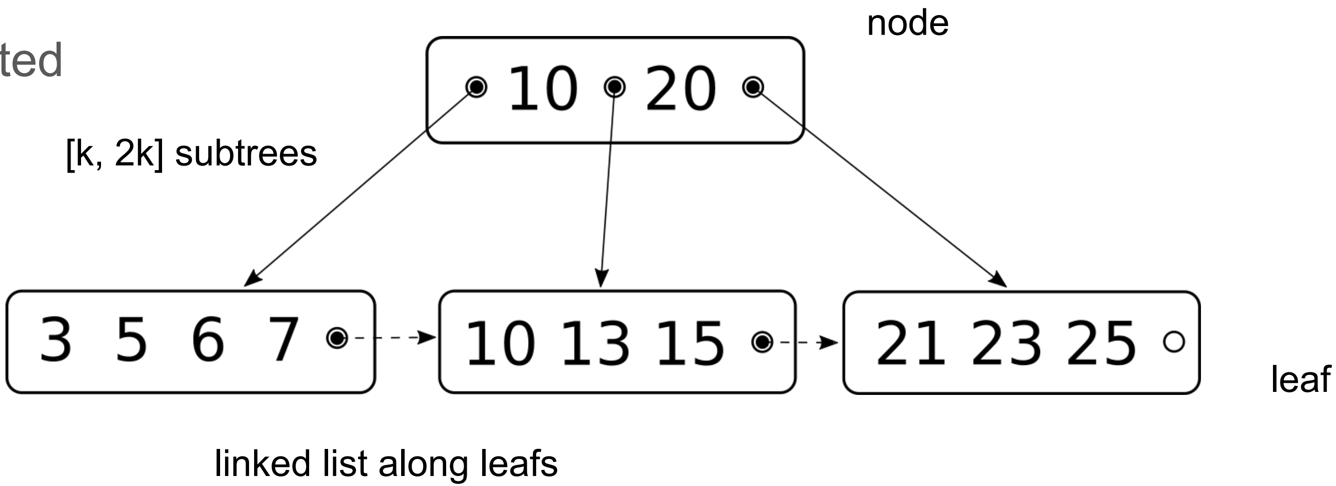
A Verified Implementation of B^+-trees in Isabelle/HOL

B$^+$-trees

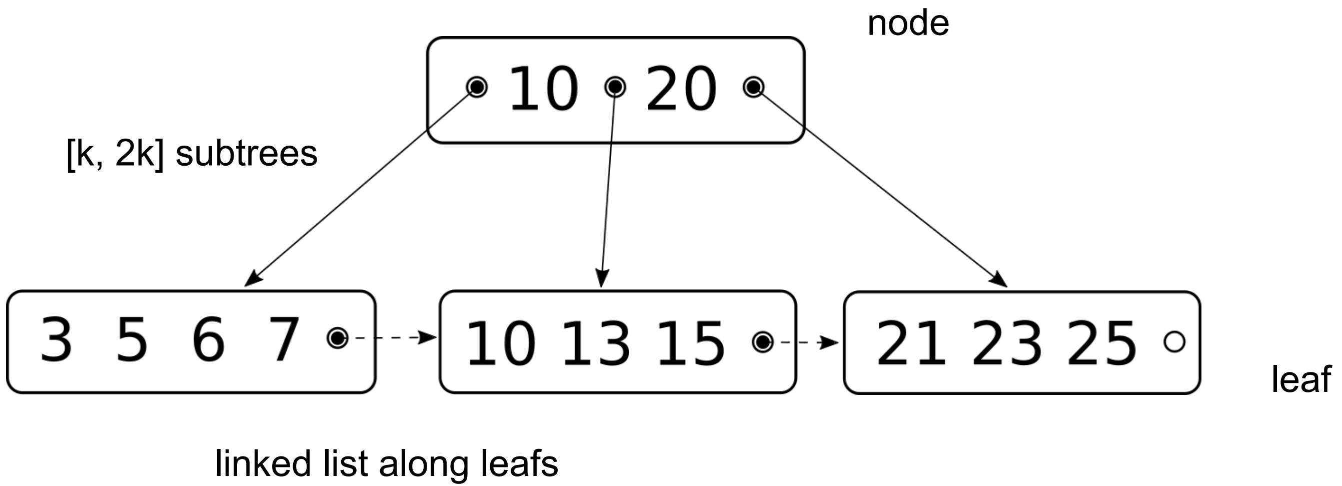
- Basis of modern database and file systems
- Ubiquitous and safety critical
- Non-trivial analysis

- balanced, sorted



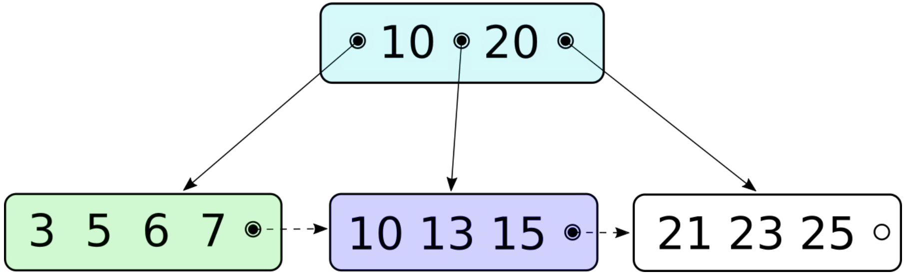
Verified Implementation of B$^+$-trees

- Verification of
  - isin/insert/delete
  - iterators
  - range queries
- Obtain executable imperative implementation



Refinement Approach

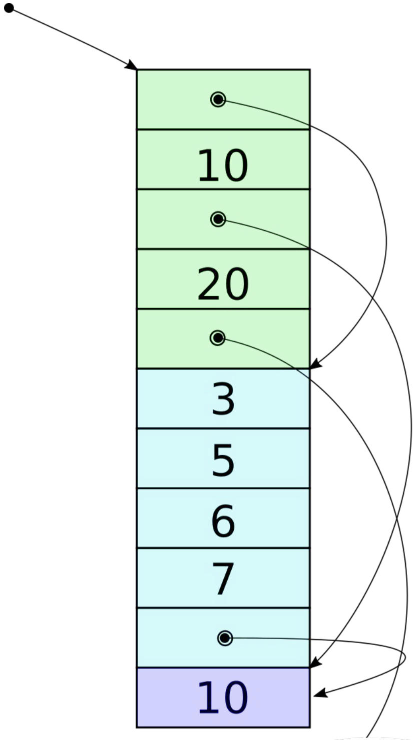
Functional Definition



- abstract reasoning
- no pointers

Refinement

Imperative Refinement



Refinement Approach

Functional Definition

```
datatype ‘a tree =
   Leaf (‘a list) |
   Node ((‘a tree × ‘a) list)
        (‘a tree)
```

→

Imperative Refinement

```
datatype ‘a tree_i =
   Leaf_i (‘a array) (‘a tree_i ref option) |
   Node_i ((‘a tree_i ref option × ‘a) array)
          (‘a tree_i ref)
```

Refinement Approach

Functional Definition

datatype ‘a tree =
  Leaf (‘a list) |
  Node ((‘a tree × ‘a) list)
    (‘a tree)

Imperative Refinement

datatype ‘a tree_i =
  Leaf_i (‘a array) (‘a tree_i ref option) |
  Node_i ((‘a tree_i ref option × ‘a) array)
    (‘a tree_i ref)

Refinement relation: $t \sim_t t_i$: ‘a tree ⇒ ‘a tree_i ⇒ bool

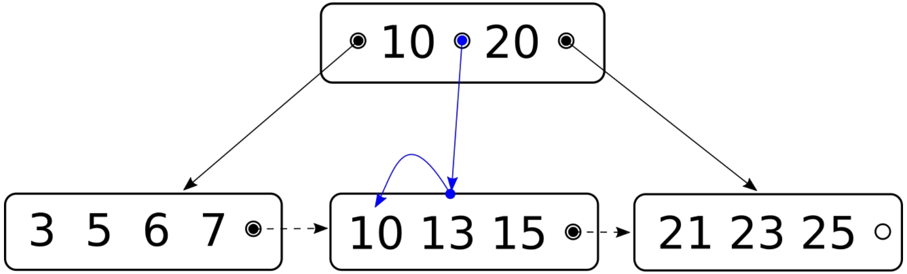
- $(\mathrm{Node}\ xs) \sim_t (\mathrm{Node}_i\ xs_i) =$
  $\exists xs_{int}.\ xs_{int} \sim_A xs_i * \forall((k,x),\ (k_i,x_i)) \in \mathrm{zip}(xs,\ xs_{int}): k = k_i \land x \sim_t x_i$
- $(\mathrm{Leaf}\ ks) \sim_t (\mathrm{Leaf}_i\ ks_i\ \_) = ks \sim_A ks_i$
- $\_ \sim_t \_ = \mathrm{false}$

Functional definition

```
fun isin: ‘a tree ⇒ ‘a ⇒ bool where
   isin (Leaf ks) x = isin₁ x ks |
   isin (Node xs t) x = case split xs x of
       (_, (sub, sep)#rs) ⇒ isin sub x |
       (_, []) ⇒ isin t x
```

assume: sorted(t)

show: x ∈ set(t) ↔isin(t, x)



Refinement Approach

Functional Definition

```
fun isin: 'a tree ⇒ 'a ⇒ bool where
   isin (Leaf ks) x = isin_l x ks |
   isin (Node xs t) x =
     case split xs x of
       (_, (sub, sep)#rs) ⇒ isin sub x |
       (_, []) ⇒ isin t x
```

Imperative Refinement

```
partial_fun (heap) isin_i: 'a tree_i ref ⇒ 'a
⇒ bool Heap where
   isin_i p x = do {
      node ← !p;
      case node of {
         Leaf_i ks _ ⇒ isin_li ks x |
         Node_i xs t ⇒ do {
           i ← split_i xs t;
           xsl ← length xs;
           if i < xsl then do {
             s ← xs[i];
             let (sub, sep) = s in
               isin_i (the sub) x
            } else isin_i t x
           }
      }
 }
```

Refinement approach

assume: sorted(t), $order_k(t)$

show:

```
< t ~t ti >
isini ti x
<λ res. t ~t ti * isin(t, x) ~b res >
```

Refinement approach

assume: sorted(t), order_k(t)

show:

$$< t \sim_t t_i >$$
$$\text{isin}_i\ t_i\ x$$
$$<\lambda\ \text{res}.\ t \sim_t t_i * \text{isin}(t,\ x) \sim_b \text{res} >$$

show:

$$< t \sim_t t_i >$$
$$\text{insert}_i\ t_i\ x$$
$$<\lambda\ \text{res}.\ \text{insert}(t,\ x) \sim_t \text{res} >$$

Refinement relation

Recall $t \sim t_i$:

- $(\mathrm{Node}\ xs) \sim_t (\mathrm{Node}_i\ xs_i) =$
  $\exists xs_{int}.\ xs_{int} \sim_A xs_i * \forall((k,x), (k_i,x_i)) \in zip(xs, xs_{int}): k = k_i \land x \sim_t x_i$
- $(\mathrm{Leaf}\ ks) \sim_t (\mathrm{Leaf}_i\ ks_i\ \_) = ks \sim_A ks_i$
- $\_ \sim_t \_ = \mathrm{false}$

Recursive “view” from the root

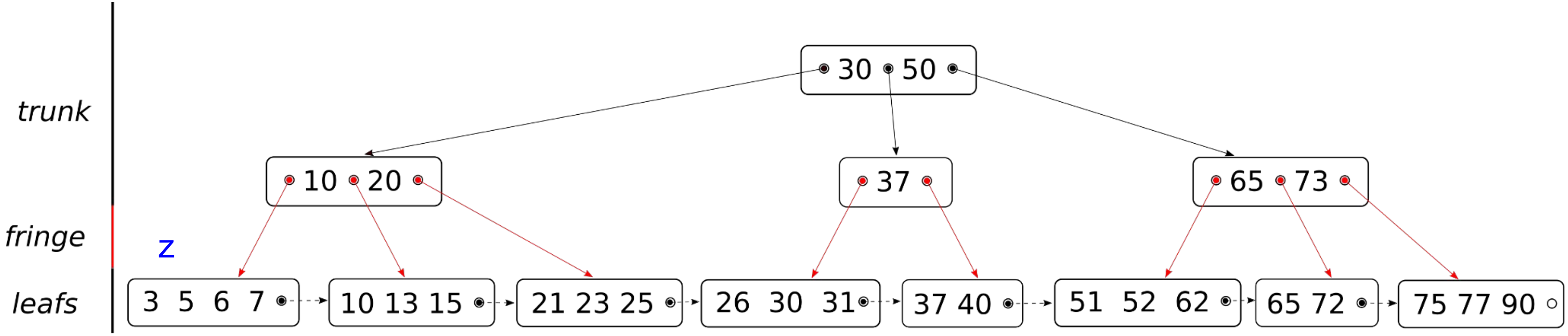
→ Need alternative when expressing iterators

Iterators

Ideally: $\texttt{t} \sim_{\texttt{t}} \texttt{t}_{\texttt{i}} \leftrightarrow \texttt{leafs(t)} \sim_{\texttt{L}} \texttt{z} * \texttt{trunk(t)} \sim_{\texttt{T}} \texttt{t}_{\texttt{i}}$

Obstacle:

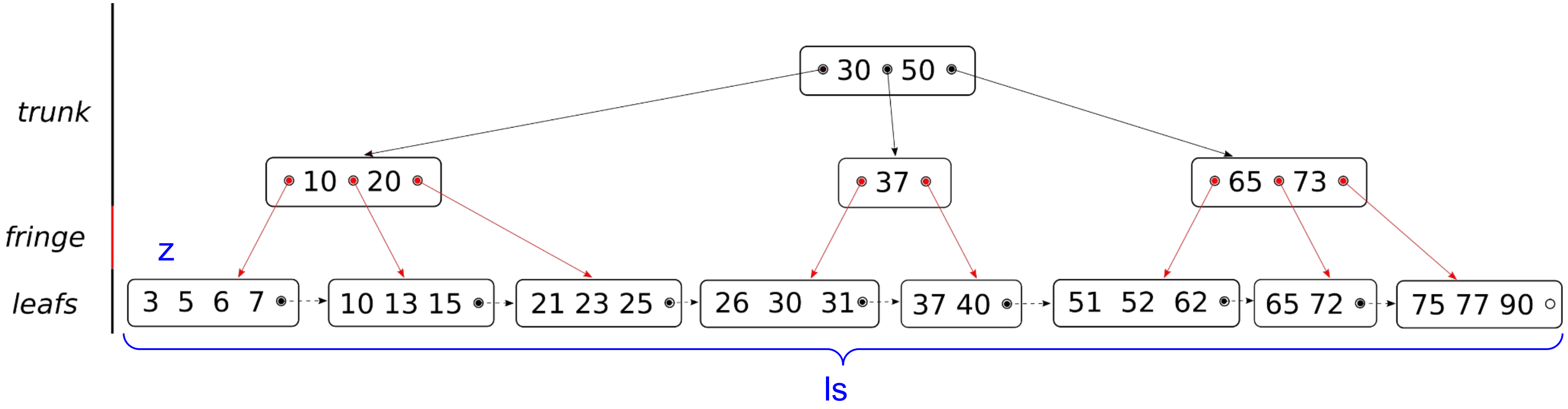
- need to show that tree can be split and re-combined



Iterators

Solution: $t \sim_t t_i \leftrightarrow \exists ls.\ leafs(t) \sim_L (z,ls) * trunk(t) \sim_T (t_i,ls)$

Ghost variable ls pins the fringe pointers



Iterators

Implement standard iterator interface from list equivalence

$$\mathtt{state}(x,z,t) = \mathtt{trunk}(t) \sim_T t_i * \underbrace{\exists xs.\ \mathtt{leafs}(t) = zs\ @\ xs * zs \sim_L z * xs \sim_L x}_{\text{current position of the iterator}}$$

1) $< t \sim_t t_i >\ \mathtt{init}(t)\ <\lambda\ z.\ \mathtt{state}(z,\ z,\ t) >$

2) $\mathtt{has\_next}(x)\mathtt{:}\ \text{‘}a\ \mathtt{tree}_i \Rightarrow \mathtt{bool}$

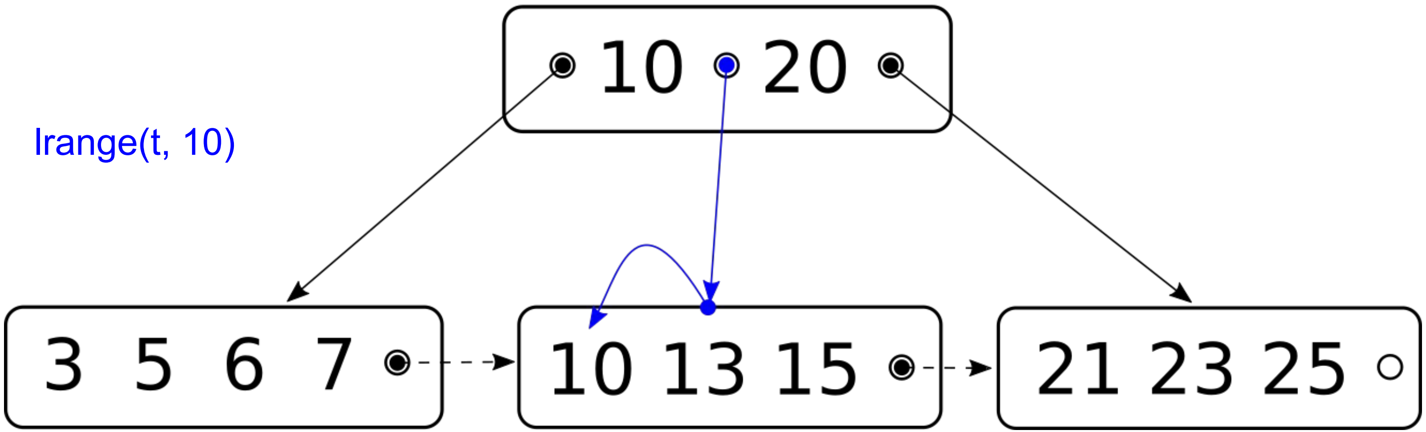
3) $\mathtt{next}(x)\mathtt{:}\ \text{‘}a\ \mathtt{tree}_i \Rightarrow (\text{‘}a,\ \text{‘}a\ \mathtt{tree}_i)$

4) $\mathtt{state}(\_,\ z,\ t) \Rightarrow t \sim_t t_i$

Range queries

`lrange(t, x) = { y | y ∈ set(t) ∧ y ≥ x }`

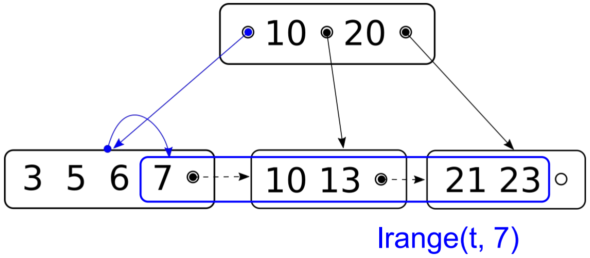
Idea: Efficiently obtain *state(x, z, t)* s.t. *next(x)* successively returns *lrange(t, x)*



Range queries

Refinement steps:

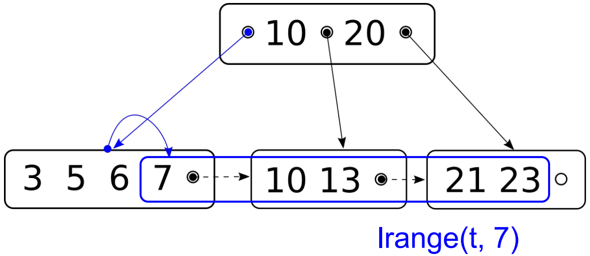
1) straightforward functional *lrange* implementation



Range queries

Refinement steps:

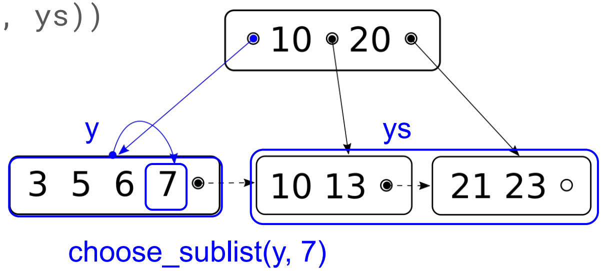
1) straightforward functional *lrange* implementation



2) *lrange_list*: collect leaf nodes and flatten later

a) `lrange_list(t, x) = y#ys : [Leaf]`

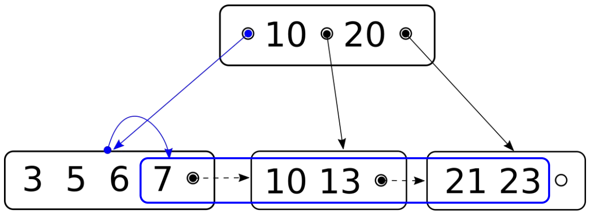
b) `lrange(t, x) = choose_sublist(y, x) @ concat(map(elements, ys))`



Range queries

Refinement steps:

1) straightforward functional *lrange* implementation

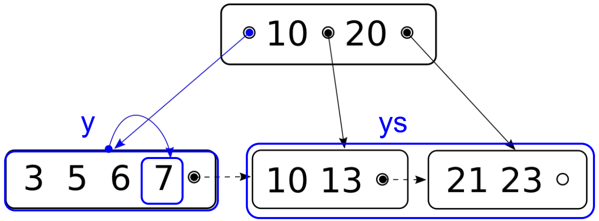


lrange(t, 7)

2) *lrange_list*: collect leaf nodes and flatten later

   a) `lrange_list(t, x) = y#ys : [Leaf]`

   b) `lrange(t, x) = choose_sublist(y, x) @ concat(map(elements, ys))`



choose_sublist(y, 7)

3) refine *lrange_list* imperatively

   a)
```
< t ~_t t_i >
lrange_list_i(t_i, x)
<λ y_i. trunk(t) ~_T t_i *
     ∃xs. leafs(t) = xs @ lrange_list(t, x) *
          xs ~_L z * lrange_list(t, x) ~_L y_i >
```

   b) transform post condition to desired iterator state (see iterator state def.)

Related work

- Pen and paper by Fielding [TR 1980] and Sexton and Thielecke [MFPS 2008]
  - gradual refinement
  - separation logic
- Ernst *et al.* (KIV/TVLA) [SSM 2015]
  - direct imperative verification
  - improved shape analysis, without linked leafs
- Malecha *et al.* (Coq) [POPL 2010]
  - tree including linked leafs, iterator
  - refinement based

Evaluation

- strict separation of functional and imperative layers
- modular design, allowing for substitution with more efficient implementations

	Malecha *et.al.*[+]	Ernst *et.al.*[d]	Our approach[+]
Functional	360	-	413
Imperative	510	1862	1093
Proofs	5190	350 + 510 + 2940	6432 + 2231
Time (months)		6+	6 + 6

[+] implementation of iterators, [d] implementation of deletion

Conclusion

- formal verification of imperative B^+-trees succeeded
- implemented and verified first efficient range query
- including binary search for node internal navigation